TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

MASTERARBEIT

Dynamic Binary Translation for Automatically Generated Simulators

ausgeführt am Institut für

Computersprachen

der Technischen Universität Wien

unter der Anleitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

durch

DAVID RIGLER

Defreggerstrasse 4E/1

3100 St. Pölten

Wien, am 30. September 2008

Abstract

The development effort for an application specific processor design with an associated software toolchain can be considerably reduced by the use of automatic tool generators. The idea of a universal processor architecture description language as basis for the automation of as many as possible parts of the toolchain, seems to be a promising approach in this direction.

The language xADL is able to universally describe architectures on a structural level and was especially developed for the embedded systems field. The xADL framework has a modular extendable design: various modules share a common preprocessed architecture description to generate different parts of the software toolchain.

In this work an xADL module that generates processor simulators along with applied basic concepts is presented. In the main part the technique of dynamic binary translation to improve the performance of those simulators is discussed and the LLVM framework, which is employed for this purpose is briefly described. Several methods to further increase simulation speed are presented and their effects are finally evaluated by means of simulators for two exemplary architectures.

Kurzfassung

Die Erstellung eines anwendungsspezifischen Prozessordesigns inklusive passender Entwicklungswerkzeuge kann durch den Einsatz von Generatoren erheblich erleichtert werden. Ein Ansatz in diese Richtung ist der Einsatz einer universellen Architekturbeschreibungssprache, auf deren Grundlage möglichst viele Teile der Entwicklungswerkzeuge automatisch erstellt werden können.

Die Sprache xADL wurde speziell für den Embedded-Systems Bereich entwickelt und ermöglicht eine universelle Beschreibung auf einer strukturellen Ebene. Das zugehörige Framework ist erweiterbar und modular im Aufbau: verschiedene Module teilen sich eine vom Framework aufbereitete Architekturbeschreibung, um die unterschiedlichen Softwareteile zu generieren.

Die vorliegende Arbeit stellt ein xADL Modul zur Generierung von Simulatoren vor und erklärt die dafür verwendeten Konzepte. Im Hauptteil wird die dynamische Übersetzung von Maschinencode zur Beschleunigung dieser Simulatoren diskutiert und auf das dazu eingesetzte LLVM-Framework kurz eingegangen. Diverse Ansätze zur weiteren Steigerung der Performance werden präsentiert und in einem abschließenden Vergleich anhand zweier konkreter Beispiel-Architekturen bewertet.

Contents

Introduction

The Institute of Computer Languages at the TU Vienna has initiated the development of a generic architecture description language called xADL. It originates in the need for optimising retargetable compilers for application-specific embedded processors and revealed to be generic enough for the automatic generation of a wide range of hard- and software components like processor simulators or even hardware descriptions (VHDL). Besides the definition of the xADL language, an extensible framework is provided that comprises a variety of modules. These modules analyse a described architecture and generate specific parts of its toolchain.

In the course of two master theses a simulator generator module was implemented. Besides the development of a generator, producing retargetable simulators with instruction decoders, techniques of dynamic binary translation were applied to speed up these simulators. The design and implementation of the simulator generator and its optimisations were done in close collaboration of the authors. As two separate theses were intended the presentation of the overall project is divided into two individual parts: While one explains all necessary steps needed to translate an xADL description to a full-blown interpreting simulator, the other one focuses on applying dynamic binary translation in the context of these retargetable simulators.

As the fundamentals of the two theses have the same origin, the first chapters (including parts of this introduction) were authored in close cooperation and are identical in both works. At the beginning of the shared part various aspects of computer architectures to be considered for description languages and simulations are summarised (with a focus on embedded systems). A choice of description languages is categorised and compared to the philosophy of xADL. Its syntax and semantic are briefly defined and language constructs are outlined. The last shared chapter describes the simulation framework (those parts of the simulator not being generated) and roughly explains an internal description used as the basis for the two different parts of generated simulators: the interpreter and the dynamic binary translator.

This thesis describes the concepts of the applied dynamic binary translation. After the shared chapters, related work regarding runtime translation of binary code is summarised. An introduction to the LLVM compiler framework, which was used as a substructure for runtime code compilation is presented next. The main part explains dynamic binary translation in detail and how it was deployed and refined to obtain high efficiency in the scenario of retargetable simulators. Starting at the compilation of basic blocks, considerations about increasing performance are made, which eventually lead to the concept of hot code identification and the "trace" as compilation unit. Additionally caching of compiled code and the aspects of simulation at the processor pipeline level are discussed. Special optimisations that are used to overcome shortcomings of LLVM finalise this part.

In a detailed evaluation two example architectures are used to sumarises the efficiency of the proposed translation technique for automatically generated simulators and its performance is compared to simulation using pure interpretion. The impact of different parameters on translation behaviour is investigated and forms the end of this diploma thesis.

This work is supported in part by ON DEMAND Microelectronics and the Christian Doppler Forschungsgesellschaft.

Chapter 1

Computer Architectures

It may be helpful to have definitions at hand when introducing abstract concepts like computer architecture. But as both terms “computer” and “architecture” cover wide areas in culture, science and everyday life such an exact definition can hardly be stated without some aspects getting out of sight.

To approach computer architectures let us first ask for the purpose of architecture in terms of planning and constructing buildings. Buildings serve fundamental and practical human needs like protection against natural environment as well as aesthetical ones. In practice there are a variety of subtle needs, goals and constraints a building has to meet. Without getting too philosophical this view clearly focuses on material aspects - the intended end product.

A similar view can be put on micro processor architecture. For one of the biggest sectors of the computer market - the desktop computer - there is a demand for the highest performance available per cost unit. This manifests in the end product: Small feature sizes provide maximal transistor counts, allowing expensive (in terms of transistors) performance optimisations at reasonable price.

When looking at a computer architecture, one can ask, what the reasons for design decisions might have been and find explanations for them - but immediately the question arises if one can think of a better implementation. This leads to another view: Architecture can be seen as the translation of abstract ideas to a concrete plan. A plan that most likely (but sometimes only hopefully) covers all needs and respects all constraints. Chapter 2 will cover architecture description languages that account for this view and allow to formalise, to plan computer architectures.

1.1 Instruction Set Architectures

A look at the computer's history reveals that computer architecture first was a synonym for instruction set architecture. The instruction set architecture defines the number and types of operations, registers and memory addressing modes to be covered by the instruction set. The need for convenient assembler programming, maximal instruction execution frequency and minimal instruction count was initially seen as a problem of designing optimal instructions. It revealed that rapidly growing register budgets allowed on-chip dynamic translation of instruction sets - the structural implementation of computer (micro)architectures began to play the key role.

Nevertheless the instruction set architecture is important, as it is the interface to the micro architecture - still an instruction set may be favoured over another depending on how computer resources are accessed. Scientific computing may favour instruction sets with special operations while for complicated string operations rich addressing modes and conversion operations may be interesting. Recent instruction sets feature subsets that can be dynamically switched between (e.g. arm thumb) or high parallelism to be solved at compile time (VLIW architectures).

The following sections are a rough overview and mainly summarise [Hennessy and Patterson, 2007].

1.1.1 Categorisation

Instruction set architectures differ in how operands and results are treated - a first categorisation of instruction sets pays attention to these differences. Typically sources for operands and destinations for results are organised as register files, linear memories or stacks.

Stack: All operands are taken from the top of a memory stack. The result is pushed back on top again.

Accumulator: One special register (the accumulator) is implicitly taken as one operand. The result of an operation is stored back to the accumulator again.

Memory/Register: Operands of operations can be either taken from memory or a register file. The result may be stored back to a register or a memory location.

Register/Register Load/Store: Operands must be taken from register files, the result must be stored back to a register. Access to memory is realised solely by special load/store instructions.

1.1.2 Operations

The main purpose of an instruction set is to define the operations a micro architecture can perform per instruction cycle. One categorisation divides instructions into data manipulation instructions and control flow instructions.

Finer differentiation of data manipulation instructions divide them further into arithmetic and logical, data transfer, floating point, decimal and string manipulation instructions. Every field of operation may favour some special purpose operations like multiply-add for signal processing, powerful bit manipulation for control systems or high precision floating point for scientific calculations.

It is worth to take a closer look at control flow instructions, as they are some of the most frequently executed instructions, terminate basic blocks and need special handling in simulators that use binary translation. In general they can be divided into three categories (compare with [Hennessy and Patterson, 2007]):

Conditional Branches: Change programflow under certain conditions

Jumps: Unconditionally change programflow

Calls and Returns: Change programflow, save/restore the program counter

Control flow instructions (called jumps for simplification) must define how the program counter is modified. Either it is directly set to an absolute value (register value, value in memory, etc.) or it is modified by adding a displacement, which is called PC-relative. PC-relative addressing saves bits for short distance jumps and makes code position-independent. Absolute jumps are used for constant jump targets like long distance calls but also when jumping to calculated addresses (typically called computed jump) like dynamically loaded library functions, virtual functions and branches of a switch statement. Calculated jumps play a key role in simulators as they cannot be easily evaluated before runtime. It should be noticed, that the return instruction is a special form of a calculated jump.

Theoretically every state within the microarchitecture is a candidate to be used as a condition for branches. In practice either a single register value, the relation between two registers, a single flag or the logical combination of flags determine if a branch is taken or not.

Some instruction set architectures (like ARM) define conditional execution not exclusively for branch instructions. Another extension to conditional execution are speculative instructions. These can be used to give the microarchitecture hints, if a branch is likely to be taken or to prefetch data (speculative loads). Instruction set architectures that are designed for low power consumption introduce even more special techniques. Some of these are: loads and stores that bypass the tag memory of a cache (a hit must be statically known), a register file hierarchy, energy-reduced instruction alternatives and explicit exception state management [Asanovic, 2000].

1.1.3 Operands

Operations can be seen as functions taking one or more input values producing (after a finite amount of microarchitecture cycles) one or more output values. The categorisation of different sources and destinations will be fuzzy, as theoretically every architectural state may be a source operand or a destination for a result:

- Registers, often specified by their index within a register file
- Internal state, often defined by a name (e.g. flags)
- Constant values, immediate values (also called constants and immediates)
- Memory objects, specified by their address

Section 1.1.4 describes handling of memory objects. Subsequently some aspects of register file access will (not exhaustively) be mentioned. First, registers may be divided into several banks. Each bank may be used by a subset of instructions or banks may act as shadow registers that can be dynamically switched. Partitioning to form register file hierarchies pays attention to the principle of locality (like caches do). Finally subregister naming allows instructions to access parts of a register by a special name (e.g. within the x86 instruction set).

1.1.4 Memory Access

An instruction set architecture must define how memory is accessed. Theoretically every method that brings bits onto the data/address bus of a memory is relevant. But with description languages in mind only the most common and important aspects of memory accesses should be summarised here.

Byte order is a first aspect differing from machine to machine. Byte order is relevant whenever multiple bytes are read/written to form larger entities (e.g. 32 Bit words). If a 4 Byte word is written and read again, the internal order of bytes is irrelevant, but if the same word is read with single byte accesses the order does matter. More precise: It defines how small portions of memory are aligned within larger ones.

The two common byte orders are called little endian and big endian. Little endian puts the least significant byte to the lowest byte address and the following ones to ascending addresses, while big endian places bytes exactly reverse. Order can also be relevant for smaller portions of memory and even single bits, but smaller than byte sized memory accesses are uncommon, single bits are usually masked out of larger entities.

Data transfer sizes within an instruction set are typically the size of the data bus and multiples/fractions of it. The MIPS architecture for example has a 32 Bit wide data bus and allows 8, 16, 32 and for special cases 64 Bit wide objects to be accessed within one instruction. An access to an object of size S at address A is called aligned if A mod S = 0. Some architectures do not allow unaligned accesses, as they might impose multiple accesses to memory.

Another aspect of memory usage within the instruction set is the calculation of memory addresses. Typically the address used by an instruction is composed of a base address and a displacement (which is added to the base). The base typically is a register value (represented in the instruction by a register index) and the displacement may be a register or an immediate value. Techniques like memory management or usage of multiple memories lead to more complicated memory addressing.

Furthermore a memory access may not only fetch data but also increase registers (by a displacement or a constant), make physical out of virtual addresses, throw exceptions caused by violation of permissions or ranges and finally may reorder bytes. Thus memory accesses are not trivial to simulate, which is also true for micro architecture in general as methods to overcome the gap between memory bandwidth and latency (e.g. caches) cause high complexity.

1.2 Computer Micro Architecture

The micro architecture of a processor deals with the processors internal structure. It mainly provides a correct implementation of the instruction set making technical details like access to a cache, branch prediction or address calculations transparent. Of course these details are not fully hidden, they manifest (for example) in bounds for instruction execution times.

For a simulator this means that the more accuracy is required, the more aspects of the micro architecture have to be simulated. Without going into too much details this chapter presents those structures and methods used by micro architectures influencing cycle counts - targeting structures to be simulated by cycle accurate simulators. The intended use of the simulator for embedded systems further narrows down the topics.

1.2.1 Pipelining

"Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast CPUs" [Hennessy and Patterson, 2007].

Each instruction is divided into operations to be handled at each pipeline stage. For a given instruction set this division is not immediately clear. The longest delay of a stage determines the speed of the whole pipeline and pipeline hazards may occur. On the other hand pipeline design may influence instruction set decisions.

Hazards are the danger of influencing the instruction set in a way that was not intended: the danger of producing wrong results. They result from the nature of pipelined execution and are typically divided into three categories.

Structural Hazards: A resource is needed in n different stages simultaneously - only less than n accesses are possible.

Data Hazards: Two instructions are in the pipeline. Read after Write (RAW) also called true dependency: the result of one instruction is needed by another one in the pipeline. Write after Write (WAW) and Write after Read (WAR), also called anti dependencies: writebacks to the register file may be out of order and cause wrong register file content (at some time).

Control Hazards: The program counter is changed - some instructions not being intended to get executed are in the pipeline.

One way to overcome pipeline hazards without introducing new hardware structures is to simply forbid instruction sequences causing them - the compiler must rearrange instructions or insert instructions that do not modify the architectural state (no-operations or nops) to overcome hazards. Below other methods are presented that do have impact to hardware structures in micro architectures, each addressing one or more types of hazards.

1.2.2 Stalling

A prominent structural hazard occurs in Von Neumann architectures: the simultaneous access of the memory by the instruction fetch unit and some other unit needing an operand from memory. A solution is to stall the pipeline: The fetch stage neither touches the memory nor increases the program counter during that cycle - it produces a bubble (nop instruction) instead. The bubble resolves the conflict but consumes one extra cycle (the real fetch is delayed).

More general: If the hazard caused by two stages with numbers i and j (let $i < j$) is to be resolved by stalling, stage i delays its conflicting operation until the conflict is resolved. Every delayed cycle a nop is passed to the successor stage and all stages $< i$ are blocked.

1.2.3 Forwarding

Data hazards may also be solved by stalling, but it is more efficient (in terms of cycles needed) to directly connect the output of one unit producing the result to the input of another unit needing it as an operand. For example the result of the ALU can be forwarded to both inputs of the ALU itself. Two criteria must apply to make use of the forwarded value: The destination register index for that value must be the same as for the input value and the forwarded value must be valid for that cycle. Thus only the needed register value (no future or former value) is forwarded.

1.2.4 Caching

A cache together with stalling can also resolve structural hazards. It is a special form of memory duplication: It holds a subset of the values of the cached memory - if a value is present in the cache it can be fetched from there, while another pipeline stage can use the main memory or another cache of it. Nonetheless the pipeline has to be stalled if a cache miss occurs. Of course the main purpose of a cache is to overcome high memory latencies and not to resolve pipeline hazards.

1.2.5 Branch Prediction

The fetch unit has to know whether a conditional branch was taken and to which address control flow was transferred to. Within a pipeline this information may not be available in time: Instructions following the branch may be in the pipeline although the branch is taken. Depending on the instruction set architecture these instructions are either executed as so-called delay slots or have to be replaced by nop instructions (pipeline flush). For example the MIPS R4000 branch instruction consumes 3 cycles with one of them used as a delay slot. Branch predictors are state machines that “guess” the outcome of a branch to fetch instructions likely to be in the pipeline after a branch and thus try to avoid CPU cycles consumed by wrongly fetched instructions. A well designed fetch unit will correctly predict jumps at a high rate.

1.2.6 Advanced Pipelining

A microarchitecture may contain more than one pipeline (e.g. in case of VLIW architectures) and a pipeline may split up to allow parallel execution and then reunion again for writing results.

The assignment of instructions to different successor stages is called issue. An exemplary implementation of Tomasulo's algorithm and extensions to it (solving problems involved in instruction scheduling (e.g. imprecise interrupts) are presented in the illuminating article [Sohi, 1990]. VLIW architectures have one fetch unit and multiple decode units but issuing may be fixed - therefore the multiple pipelines are called slots.

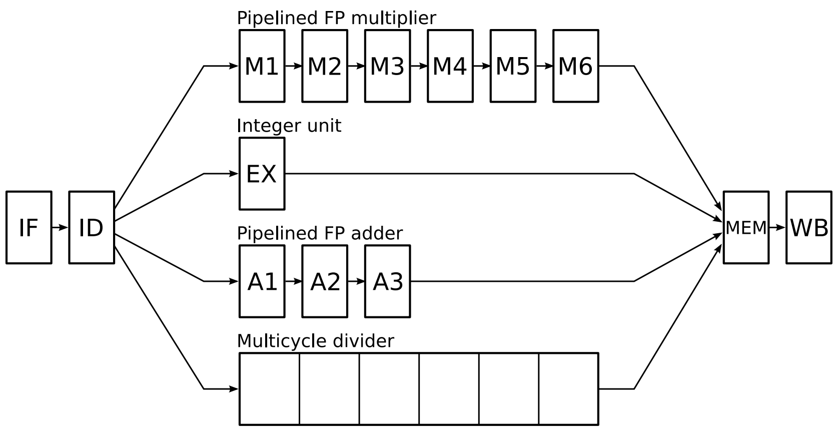


Figure 1.1: An advanced MIPS pipeline

Figure 1.1 illustrates an advanced MIPS pipeline presented in [Hennessy and Patterson, 2007]. It should be noticed that subpipelines may have different length and therefore instructions may reach the stage that merges the pipelines out of order - simultaneous writebacks and WAW hazards may occur. To avoid such hazards the issue stage must stall parts of the pipeline in scenarios detected by a stall logic. Reservation tables can be used by compilers to avoid stalls or by cycle accurate simulators to account for them. The advanced MIPS pipeline enables parallel execution of instructions, however every cycle at most one instruction can be issued. In contrast so-called superscalar architectures are able to issue multiple instructions each cycle - they reach a clock count per instruction smaller than one.

VLIW can be seen as a simplification of superscalar architectures - instruction scheduling is no longer done in hardware at runtime, but instead by the compiler at compile time. Multiple already scheduled instructions are fetched simultaneously - each filling a slot and being executed in a dedicated pipeline. The obvious advantage of this concept is its (hardware architectural) simplicity with the downside of increased instruction bandwidth and the dependency on high quality compiler designs.

Other advanced topics include dynamic scheduling resulting in out-of-order issue, execute and writeback (e.g. CDC 6600 processor with scoreboards) or register renaming (solving WAR and WAW hazards) - But these topics go beyond the scope of this work.

Chapter 2

Architecture Description Languages

Advances in microchip technology lead to integration densities that allow whole systems to be implemented on a single die. These system-on-chip (SoC) systems offer interesting opportunities like shorter interconnects between subsystems and thus higher bandwidths or composition of many subsystems that are optimised for the specific application. Designs can profit from such application specific optimisations (e.g. in terms of performance or energy efficiency) but they have the drawback of longer design and implementation phases (compared to commercial of-the-shelf products) resulting in higher non-recurring engineering (NRE) cost.

Especially application specific processors are challenging as design decisions not only affect the hardware itself but also tools that are needed within the implementation process of the application software like compilers, assemblers or simulators. In the phase of design space exploration these tools are typically implemented and evaluated in an iterative process. Processor architecture description languages that support automatic generation of tools have the potential to dramatically shorten this process of design space exploration.

The term architecture description language covers hardware and software architectures but will be used as a synonym for hardware architecture description languages within this work.

2.1 Classification

Several processor description languages have been proposed, many of them originate in the demand for retargetable compilation infrastructures (see [Qin and Malik, 2002]). A first categorisation can be made in respect of the level of abstraction used. The higher the abstraction of a language the more difficult the task to cover a wide range of processors - very specific or uncommon features may not be covered by the semantics - the lower the level of abstraction the more details must be explicitly stated, resulting in longer, more error-prone and harder to read and understand descriptions.

Hardware description languages such as VHDL or Verilog allow descriptions to be very low level - for example they allow for strongly asynchronous designs but lack sufficient abstraction to explore architectures at the system level [Mishra, 2005]. Modelling languages like UML on the other hand put focus on the whole system, describing subsystems and interfaces but leave open how components of subsystems function.

A quite common level of abstraction for architecture descriptions is that of register transfer level or RTL. It is characterised by using the clock cycle as the granularity of time, which is the time between two consecutive clock events on the physical hardware. Furthermore dataflow is modelled to happen after each clock cycle (at a clock event) by moving data from the input side of a register to its output side (infinitely fast). Data manipulation operations happen during one (or more) clock cycles. For synchronous systems this representation is very intuitive and has a long tradition in digital design.

Besides different levels of abstraction a distinction can be drawn between structural and behavioural descriptions, where in fact architecture description languages will contain both with putting focus on one of them. The semantics of a "structural" language together with a given description (e.g. a netlist) imply possible behaviours of the system. A "behavioural" description together with language semantics imply possible (possibly a lot of) structures that realise that behaviour. Even classical programming languages could be used to model hardware solely behavioural, but as they support a very high level of generality every single detail would have to be hand-coded.

2.2 Related Work

2.2.1 MIMOLA

MIMOLA is a language developed in the late 70ies at the University of Kiel. With MIMOLA hardware descriptions at RT-level can be generated out of descriptions of typical applications (in a Pascal-like form) together with execution frequencies of paths through these applications and replacement rules for application elements [Mishra and Dutt, 2004]. Besides MIMOLA allows predefined resource specifications that can partly or fully describe a machine structure (through modules and their interconnects at a RT-level). Missing structures are generated by MIMOLA as a result of transforming different MIMOLA language inputs - MIMOLA is therefore categorised as a structural description language [Mishra, 2005].

2.2.2 ISDL

With ISDL (Instruction Set Description Language) an architecture is described on a behavioural level. It is designed to cover a broad range of instruction set types like VLIWs or DSP extensions and serves as a starting point for generating a compiler and an assembler [Hadjiyiannis et al., 1997]. Generality is reached through the use of C syntax wrapped into different language constructs that form several sections:

- Instruction Word Format: Names are provided for bitfields (i.e. consecutive bits).
- Global Definitions: Definitions in this section support the generation of Lex and Yacc [Levine et al., 1992]. So-called split functions allow long bit sequences (like immediate values) to be split up into multiple bitfields.
- Storage Resources: Register files, single registers and memories can be defined.
- Assembly Syntax: Parallel operations for each single instruction are defined.
- Constraints: Reduces the number of possible combinations of operations to those allowed by the architecture.
- Optimisations: This section is optional and intended to give compiler hints resulting in faster code.

Later achievements [Hadjiyiannis et al., 1999] include an instruction set simulator and present algorithms for extraction of structural information that can be used by a hardware synthesis tool. The generation of the simulator is not described in detail.

2.2.3 Expression

Expression is a description language that consists of behavioural and structural elements. Its main goal is the support of design space exploration and software toolkit generation. An interesting feature of Expression is its ability for rapid design space exploration that allows the designer to make use of reduced accuracy of descriptions resulting in coarse evaluation of alternative design candidates. With the full description it is possible to generate an optimising compiler and a cycle accurate simulator. Scheduling information for the compiler that is caused by conflicts in the pipeline is not explicitly specified in the description but computed out of the structural description. Different memory systems (including hierarchies and buses) can be specified in expression and are respected by the generated optimising compiler. The language itself consists of different sections that specify operations, instructions, components, data transfer paths and memory systems. The use of expression for automated generation of VHDL hardware descriptions is presented in [Mishra et al., 2004].

2.2.4 LISA

LISA is an architecture description language initially designed to enable retargeting of fast compiled simulators [Pees et al., 1999]. A LISA description allows to specify pipelined architectures on an operational level enabling the generated simulator to be cycle and even phase accurate. The behavioural description of one instruction is built up out of atomic operations (e.g. arithmetic or logical). Precise timing information on the other hand result from resource consumption expressed by atomic operations organised in extended Gantt charts. A restriction of LISA is the assumption that bubbles are inserted in the pipeline on resource conflicts, thus disabling simulation of out-of-order architectures. An extension to LISA is described in [Braun et al., 2004]. The generation of compilers was enabled by adding special semantics code to the description.

2.2.5 nML

nML is a mixed description language that embodies structural an behavioural descriptions. Storages (memories) are the central elements being both used as a skeleton for structural interconnects and as the representation of the architectural state [Fauth et al., 1995].The programmers view of the architecture (including assembler mnemonics and binary encoding) is used as the level of abstraction leaving details of the micro architectures implementation undefined. Descriptions in nML are however rich enough to be used as models for automatically generated simulators [Rajesh and Moona, 1999].

2.2.6 MADL

The Mescal Architecture Description Language (MADL) described by [Qin et al., 2004] has been used to generate cycle accurate simulators, instruction schedulers and register allocators. The layered approach partitions toolset specific constructs and architecture specific functionality. A core ability of MADL is the possibility to define instructions via finite state machines. Structural and data resources are modelled as tokens passed to these state machines. Functional units can act as token managers assigning hardware resources to instructions. This describes parallelism in a natural way and covers more complex architectures like modern superscalar processors with out-of-order execution. MADL was used to generate cycle accurate simulators for different architectures.

2.2.7 PD-XML

Seng et al. propose a language with XML syntax [Seng et al., 2002]. A description consists of three types of elements: storage (e.g. register files), an instruction set description and a resource (microarchitecture) description. The microarchitecture is described on high or low level, where one high level description can be mapped to different low level implementations - supporting PD-XMLs claim for high flexibility.

2.2.8 ArchC

ArchC [Azevedo et al., 2005] is a description language based on SystemC with a higher abstraction level intended for processor description. The two parts of the description define the architecture resources on one hand and the instruction set on the other. The possibility to verify detailed descriptions with coarse behavioural models is a useful feature of ArchC. A full blown simulator generator is able to generate cycle accurate simulators that optionally use compilation techniques. An ABI section in the description defines the application binary interface used by the simulators for handling of system calls which are used to simulate an operation system environment [Bartholomeu et al., 2003]. All tools and documentations are freely available as ArchC is developed as an open source project.

2.2.9 Generic Netlists

Architecture Description Languages can be used to generate hardware descriptions as presented in [Gorjiara et al., 2006]. In this paper it is demonstrated how an application (in form of C Code) together with an XML based architecture description (generic netlist representation - GNR) can be translated to a synthesisable (and simulateable) hardware description. The architecture description only outlines the machine, while the instructions (in form of so-called control words) are generated and stored in the synthesised memories. The presented form of hardware description is targeted to very application specific processors ("tight" hardware/software co-design) but not suited to develop general purpose cores.

2.2.10 Portable Compiled Instruction-set Simulators

D'Errico and Qin present a retargetable simulator framework that uses a behavioural description to generate interpreting and translating (both static and dynamic) simulators [D'Errico and Qin, 2006]. They highlight the advantages of retargetable and portable simulators. For their translating simulators, target instructions are converted to C++ code which is compiled with GCC to one of many supported host architectures. The dynamic translating simulator uses GCC to compile dynamically loaded libraries which are loaded at runtime. To reduce compiletime only frequently executed parts of the target application are compiled - the rest is interpreted. They include measurements highlighting speed gains obtained by using run-time translation (being approximately 4 times faster).

2.3 xADL

The xADL is a structural description language developed at the TU Vienna, Institute of Computer Languages. It uses extensible markup language (XML) syntax to describe embedded pipelined architectures. They are described by their structure in form of functional units, registers and memories connected in a network. Functional units can have user defined functionality described by atomic operations that are precisely defined in xADL. Only simple memory hierarchies (without a memory mapping unit) are supported in xADL (at least at the moment) which limits its application area to processors with simple memory subsystems often found in embedded systems. It is however possible to describe different types of caches that are characterised by their features (rather than by their explicit functionality).

The xADL language is distinguished by the following paradigm: A single compact description serves as the basis for a wide variety of analyses and generators (xADL modules). Language constructs that directly support generators by giving explicit hints are avoided. Therefore many different modules can be supported without redundancies that could make architecture changes (needed in design space exploration) cumbersome or even contradicting (which could lead to inconsistent results of different modules). Examples for this paradigm are lacking explicit listings of the instruction set or reservation tables.

The compact description shifts complexity to the analyses and generator modules - a module is responsible for extracting information out of a non-specific abstract description. As mentioned above this complexity is justified by descriptions being compact and consistent.

Another interesting aspect of xADL is its support for VLIW architectures - units can simply be duplicated by defining a repeat-count.

Besides pure hardware xADL describes the interface to the software system within a special ABI section. It defines how values are passed to functions, how registers are saved on calling functions and which registers are reserved for special purposes (like the program counter and the stack pointer). This information is especially useful for compiler related generator modules but can also be important for other modules.

Currently two different architectures are described in xADL: A MIPS R2000 derivate and the CHILI architecture (a novel 4-way VLIW architecture developed by ON DEMAND Microelectronics). While the MIPS description evolved together with the development of xADL and its modules the description of the CHILI architecture could be easily implemented afterwards without any major changes to xADL. Modules that are available by now include a generator for compiler related tables and rules, an instruction set extractor, a visualisation module (producing a network graph of the architectures structure) and a simulator generator. This simulator generator, its output and all involved technologies are the topic of this work and are described in the main chapters.

2.3.1 Syntax

As already stated, XML is the base of the xADL syntax. An xADL architecture description must contain the <adl:Architecture> top level construct, an ABI tag, at least one configuration and the netlist of units and datapaths building the architectures structure.

ABI

The ABI tag specifies the binary interface used by the software system of the architecture. All registers are categorised in classes and their use for argument passing and return values is specified. Special registers like the stack pointer are defined and optionally a name is assigned to each register in the registerfiles - for example these names can be used by modules for debugging purposes or automated generation of documentation. Memories can be classified to be data and/or program memories. It is possible to define more (named) ABI sections inside one architecture description that can be switched between by executing the xADL toolkit "adlgen" with different parameters.

Configuration

It is possible to define multiple configuration tags each of which specifies values for symbolic parameters. These parameters can be used in the structured netlist description as bitwidth parameters (e.g. for registers or buses) repeat counts of units or register counts inside a registerfile. Therefore it is possible to explore different versions of an architecture with differing parameters in just one description (e.g. 32-Bit versus 64-Bit architecture). Additionally it is possible to define components inside configuration tags to enable different versions of components in a description.

Types

Before the structured netlist can be defined in xADL the types of all used components have to be declared. These components can be immediates, memories, caches, registers or functional units. The corresponding language constructs and its parameters that declare types are described in the following list.

- <MemoryType>

  Like any other type declaration memory types must be given a name. The size of the memory is specified in bytes, its latency parameters is coarsely stated in cycles by the *min-delay* and *max-delay* properties. A number of input and output ports define the memory interface. An additional address (input) port does not need to be defined - it is added implicitly for each input and output port. The bitwidth of ports and the number of bits used for addressing can be specified together with optional *baseaddress* and *alignment* attributes.

- <CacheType>

  Caches are roughly declared by their latencies (similar to memories) and a *type* attribute. For example such a *type* can be "set-associative". The exact semantics of these types is not defined in detail yet.

- <RegisterType>

  Registers have these main attributes: *name*, *bitwidth* and *repeatcount*. For this type the *repeat-count* is not only useful in the case of VLIW architectures. It is used to define register files. Ports define the interface of register files: they can be assigned a read-only, write-only or read/write attribute. The bitwidth attribute of ports defines how many bits of a register are accessed while the offset attribute specifies which bits are accessed. Note that the ports bitwidth may be different from the registers *bitwidth* and thus allows definition of subregisters (as found on x86 for example). If offset and width are omitted the full register is accessed by default. On many architectures constant registers are found. A constant tag with its *index* and *value* attributes declares a register to be constant and what value it holds.

- <ImmediateType>

  Immediates origin from instruction words. Although they are therefore not part of the hardware, they can be seen as data sources (similar to registers) and thus are connected to the network of units like any other read-only data component. The only attributes of immediate types are *name* and *bitwidth*.

- <UnitType>

  Unit types are the most comprehensive types. Like others they have a name and contain ports defining their interface. Units are the components that implement a functionality defined by their operations that themselves consist of micro operations (built-in calls). Furthermore units can encapsulate networks of other components like registers, immediates or even (sub) units to support hierarchical design - such an encapsulating unit may not define operations as they are only permitted on the lowest level of the hierarchy.

  Operations can process information delivered to input ports of units and specify how outputs are computed. It is possible to define several operations in a single unit. This means that these operations are alternative computations that can be performed by the unit. An example for this are different arithmetic operations performed in an ALU unit type.

  Besides inputs, outputs and constants, temporaries serve as additional parameters for built-in calls. They are declared inside the unit type and have *name* and *bitwidth* attributes, constants carry a value. The following section describes operations in detail.

Operations

As mentioned before operations are declared within unit types. Apart from a name operations have two specific attributes - *syntax* and *binary* which are described in Section 2.3.2. The building blocks of operations, named calls, are collected within *body* tags. Several predefined built-in calls are available to model a units behaviour. Additionally it is possible to implement user defined calls but their exact functionality and semantics are left open at the moment.

To define (complicated) operations several built-in calls can be combined - it is however not possible to nest calls. Instead temporaries are needed to reuse results as built-in call arguments. There is one distinguished built-in call to model conditional functionality: *cmove*. The functionality of the conditional move call is as follows: an argument is evaluated to true or false (like in C its value is compared to zero) and depending on that one of the two additional arguments is stored as the result of the call.

Instantiation of Types

This part of the xADL description forms up the concrete network out of instantiated components and connections. Typically it begins with instantiating the data sources (sinks) already mentioned: registers, immediates, memories and caches. These instantiations have in common to need a name tag and one to reference the instantiated type. For caches the memory instance that is cached needs to be stated. Registers have a *category* property that defines special purposes (e.g. integer, base, etc.).

However the major part of the instantiations is formed up by units. They are the only components allowed to use the *connect* tag that specifies the datapaths between components and thus the architectures structure. The syntax of these connect tags allows to specify separate *input* and *output* connections for the interface of the corresponding unit. Both of them use the *select* attribute to reference the other end of a connection (input-ports for output connections and vice versa). When two functional units are connected to each other, it does not matter whether this is described by an input connect in one unit or an output connect in the other.

An important property of connections is declared by the *stageboundary* attribute: It gives connections the semantics of transferring data only on clock events (modelling of hardware register transfer). Such connections are used to split components that belong to different pipeline stages.

Hazards have a similar syntax to connections, but quite different semantics. They are used to model (virtual) connections between units, that are used to resolve pipeline conflicts. The *forward* hazard is a connection that forwards a result from an output port to an input port and thus resolves data hazards. The *stall* hazard, if connecting an input and an output port has a similar purpose, but instead of forwarding a value it has the semantics to stall the pipeline until the result is available at the input port (e.g. until it is written back to the register file). If the *stall* hazard connects two output ports its semantics is to resolve WAW conflicts. The *ignore* hazard is a dummy signalling a hazard not being resolved in hardware (e.g. this information can be used by a compiler generating module to resolve conflicts in software).

2.3.2 Instruction Set

As already mentioned the instruction set is not defined explicitly. The instruction set is extracted by following all possible paths in the network using a breath first search. Each path represents a potential instruction of the architecture. But typically not every path corresponds to an instruction, thus xADL uses a predicate/condition concept to exclude paths. Predicates and conditions can be defined along data paths inside units, unit types and operations. The *condition* tags evaluate *predicate* tags defined earlier on the path and omit instructions that do not satisfy those conditions.

An example for such *condition* is to use a restricted set of operations in an ALU for address calculation: A predicate *addressable* could be set for addition and subtraction - an appropriate *condition* in a memory access unit could forbid all other ALU operations.

Syntax and Binary Representation

When describing an existing architecture the syntax or at least the binary encoding of instructions have to be specified to make fully automatic toolkit generation possible. In xADL the syntax of instructions is described constructively along the instruction path. In a syntax section one or more syntax templates specify different types of instructions. They may consist of strings and several tokens. Most components in the network (even connections) may use *syntax* tags that specify concrete values for tokens. Those values can be strings, values of immediates or indices in a register file.

The binary encoding of instructions is defined similarly with templates and tokens. Additionally bitmasks select characteristic bits of an instruction word. The values for these masks are assigned just like tokens along the instruction path.

This form of syntax and binary description is especially suited for orthogonal instruction sets, often found in embedded processors, because only few different syntax templates are needed.

Chapter 3

Simulator Generator

This chapter focuses on the automated generation of simulators for an architecture described in xADL. The architecture to be simulated is called “source”, while the architecture that runs the generated simulator is called “host”. There are many purposes for which simulators can be useful: (old) applications that are written for the source architecture can be used on (modern) hosts even when the sourcecode of these applications is not available or difficult to port; debugging is possible at a very low level and performance bottlenecks of applications can be detected at the microarchitectural layer where cache misses or pipeline stalls are relevant for execution speed.

When a simulator is generated from a high level description language it is easy to change important parameters of the architecture, like the number of registers or the number of functional units. To evaluate the properties of the resulting architecture, benchmarks compiled for this new architecture are needed. An automated generation of compilers from the same description language makes such benchmarks possible and enables to evaluate completely new architectures even in their (early) design phase. The xADL has the potential to provide these features and parts of them are already finished.

The simulator generator presented in this work supports the generation of both an interpreter and a dynamic binary translator. For code generation of the translator the LLVM framework is used; it provides a generic interface to produce code for a variety of host architectures and is frequently improved by its active developer community.

The core of the simulator generator is implemented in form of a module for the xADL framework. This framework (among other things) is able to extract the instruction set of a specified architecture. The simulator builds on this instruction set that specifies in detail when (in which pipeline stage) and which operations are to be performed by an instruction. This information is then transformed to an executable form and enriched by the operations needed for the simulation of the whole architecture. Besides the executable description of the instruction set, other important architecture specific parts, like the instruction decoder or the virtual register files, are generated to complete the simulator.

3.1 Simulator Framework

The source code of the simulator is not completely generated out of the architecture description language. This chapter explains the hand-written framework that contains architecture independent parts of the simulator. Important parameters for this framework like the number of pipeline stages and slots or the sizes of memories are set by pre-processor macros in the generated code.

3.1.1 Simulation Model

The core of a classical interpreting simulator consists of a loop iterating through simulated instructions of the source architecture. An instruction is loaded from the simulated program memory, decoded from its binary representation and then executed: The simulated registers and memories are altered according to the semantics of that instruction.

It is possible to implement simulation at different levels of abstraction: Instruction set simulators provide a semantic model of the instruction set but ignore the source architectures internal structure. While such simulators may perfectly run binary applications they are inherently inaccurate when exact timings (cycle count) of the simulation are needed. Especially for architectures that have memory hierarchies, multiple pipelines or other features that dynamically influence the execution speed of an application, at least some parts of the architecture's structure have to be taken into account to make cycle accurate simulations possible.

At a very low level of abstraction (e.g. transistor level simulation) internal timings even for physical phenomena like signal propagation can be taken into account.

Of course the simulation speed depends on the level of abstraction: Generally a lower abstraction level leads to lower simulation speed.

We use pipeline simulation that operates on a moderate level of abstraction and features an explicit model of the architectures pipeline structure. In that case one iteration of the simulators main loop does not simulate the semantics of a whole instruction but that of a clock cycle (shifting instructions through the pipeline).

3.1.2 Architectural State

The state of the simulated architecture consists of all its registers and memories. Besides registers that are explicitly defined in the xADL (see Section 2.3) the architectural state includes additional registers like pipeline-registers (registers between two adjacent pipeline stages) or forward-registers. While some of those registers may not even exist on the physical processors they are needed for their simulation. This is mostly because pipeline stages are executed simultaneously on a physical processor, but have to be simulated sequentially in the simulator. One aspect of this discrepancy is that physical registers might be used as input and output in the same clock cycle by different units. In many cases such simulation conflicts can be resolved by an appropriately ordered sequential execution of the corresponding pipeline stages. When no such ordering is possible additional (shadow) registers are required for the simulation.

Simulated instructions operate on the architectural state, by moving values between registers and/or modifying them according to their semantics. Each instruction can be decomposed to smaller operations and all such operations, that are performed in one pipeline stage form the so-called pipestage-function of an instruction for this pipeline stage. For each instruction and pipeline stage a separate pipestage-function is generated in form of a C-function that performs the appropriate operations on the global architectural state.

3.1.3 Pipeline Organisation

Beside the architectural state, the pipeline buffer is another important data structure used to represent the pipeline's logical state that is basically organised in form of a ringbuffer. Each entry in this buffer corresponds to an instruction currently in the pipeline. For VLIW architectures it is two-dimensional - one dimension for pipeline stages and the other one for slots.

Each entry in the pipeline buffer contains the following items:

- The type of the instruction (e.g. add immediate)

  This is technically a pointer to a generated structure that provides all stage-functions for the instruction, an instruction id and information about the instructions ability to change the programflow.

- The operands of the instruction

  These contain all register indices and immediate values of the instruction and are provided as arguments to the stage-function of instructions each time they are called.

- The program counter

It specifies the position of the instruction in the program memory and is mainly used by the dynamic binary translator to enable caching of basic blocks.

3.1.4 Pipeline Simulation

The pipeline (ring) buffer is the essential part of the simulator core loop. In each iteration a new instruction is inserted into the buffer by overwriting the oldest entry. Each instruction enters the pipeline at stage 0 and propagates to the end of the pipeline while its pipestage-functions are executed modifying the architectural state. The instructions position within the pipeline buffer determines which of its stage-functions is called.

An Example: The MIPS R2000 architecture is specified in xADL and provides an instruction "add immediate" with the internal id 69. The stage-functions generated for this instruction are therefore called "instr_69_stage0", "instr_69_stage1", "instr_69_stage2" and so on. When an "add immediate" instruction completely shifts through the pipeline (which requires 5 cycles) all its stage-functions are called one after the other - each with the operands (arguments to the C-function) specifying the immediate value to add and the source and destination register indices belonging to that instance of the "add immediate" instruction.

To simulate a full clock cycle all of the instructions currently in the pipeline have to be processed one after the other beginning at the back end of the pipeline. This ordering is chosen to overcome the need for a lot of additional registers (see Section 3.1.2).

However not all of the simulation is done in just one pass. To model special structures, like forward registers or registers that are concurrently written in different stages with different priorities, shadow registers are needed. This is because the execution order, that would be needed to avoid those shadow registers, is conflicting with the back-to-front execution order of the pipeline buffer. The contents of shadow registers have to be copied back to their corresponding origin registers (writeback) after the pipestage-functions for a full cycle are executed.

Therefore the generator has the ability to generate code for a second pass - each stage-function has its so-called epilogue that can be used for the writeback and gets executed in this pass.

In each cycle the decoder inserts a new instruction into the pipeline buffer. The next instruction to decode is determined by the source program counter (PC) which is normally used to index the simulated program memory. Jump instructions (and others that modify the programflow) change the PC in their stage-functions (or epilogues); if no such explicit modification takes place in a simulated cycle the simulator framework increments the PC by the size of the last decoded instruction - and thus the subsequent instruction gets decoded in the following cycle.

3.2 System Calls

Architectures found in embedded systems are the main target of the xADL description language. Such environments, especially for performance critical applications, sometimes lack an operating system layer between hardware and application software - interactions with peripheral hardware must be implemented directly using memory mapped I/O or special instructions then.

Nonetheless to enable a generic interaction between the simulated program and the host system, we use special system calls (syscalls) in our simulator. We provide a subset of newlib (a C library intended for use on embedded systems) syscalls. By definition the source applications initiate the execution of such syscalls by calls to special addresses. Before an instruction is decoded the current PC is compared with these addresses. In case of a match, special instructions are inserted to the pipeline buffer that contain the execution of the corresponding syscall and an instruction sequence that provides the return to the calling site of the syscall. Parameters to (and return values from) the syscalls are passed through the registers specified in the ABI section of the xADL description (see Section 2.3.1).

We use compilers configured with newlib to compile executables for our two example architectures (MIPS R2000 and CHILI) while the simulator itself is compiled with the "GNU C Library" (glibc). The two C libraries differ in naming and binary encoding of flags and parameter ordering and we have therefore implemented wrapper functions for the most important syscalls, that translate source (newlib) syscalls to host (libc) syscalls.

3.3 Binary File Loading

The simulator is capable of directly loading binary ELF (Executable and Linking Format) files, that can also be used to run on real targets. It uses the Binary File Descriptor library (libbfd) [Pesch and Osier, 1993] for this purpose and therefore an appropriate version of libbfd must be available on the host system to simulate a certain architecture. To support the loading (and execution) of binary files of architectures that are not available or still in the development phase it is (theoretically) possible to generate suitable libbfds along with the corresponding compilers out of the xADL description.

A (simplified) ELF file contains a header and a list of sections that contain binary data. To execute such files, the contents of these sections are first copied to the simulated program and data memories in the architectural state and then the simulated program counter is set to the starting address specified in the header section. Currently the simulator framework supports two separated (continuously addressed) memories for data and program sections. One of these memories is chosen depending on the type of the section to load:

sections that contain executable code go to program memory while the remaining ones are loaded to the data memory (Harvard architecture). It is however possible to define one memory for both program and data sections, which allows the simulation of Von Neumann architectures (e.g. the MIPS R2000). For more complicated memory systems, with multiple data memories and/or fragmented address space some improvements to the current implementation of the framework are needed.

3.4 Internal Description

To generate a simulator for an architecture out of a structural description it has to be transformed to a (more or less) sequentially executable form. For our work this means that all implicit and explicit semantics of the description are mapped to a sequence of executable instructions and control structures that explicitly describe how to implement the desired behaviour. For example the structure of a forward link is represented through an instruction/control sequence at each end of the link: A store instruction to a forward register at the back end and a conditional read instruction of the same forward register at the front end.

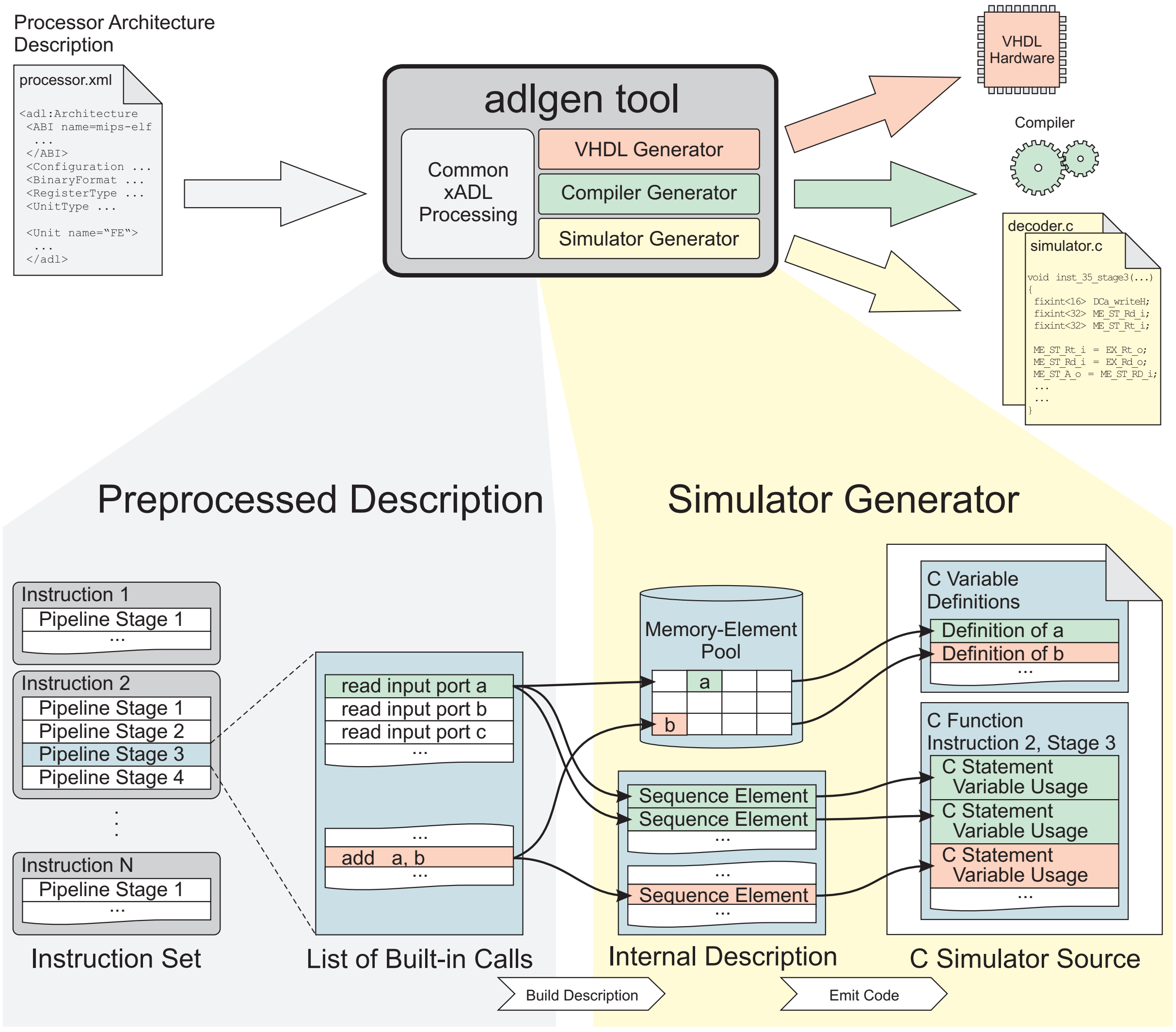
For the interpreting version of our simulator these instruction sequences are generated as C-code, that directly operates on the architectural state which is implemented in form of global C variables. In a first approach the generation of C-code out of the description was made more or less directly in one pass, printing one or more C-statements for each operation along the extracted instruction path (see Section 2.3.2 and [Brandner et al., 2007] for instruction path extraction). The simulator generated that way worked but needed a lot of special cases in the generator to cope with all the semantics possible in the xADL description (and even did not cover all of them). But the main disadvantage of this first version was its lack of extendability.

Our goal was to extend the purely interpreting simulator to a (partly) translating one. The idea was to generate C-code that instead of directly modifying the architectural state, would itself generate (machine) code that performs these modifications. To achieve this goal we use the LLVM framework, which offers a C++ interface to generate the needed machine code [Lattner and Adve, 2004].

So the simulator generator needs to generate two forms of executable sequences, which have completely different appearance but very similar structural composition. To accomplish this, without duplicating big parts of the generator code, we introduced an internal description of (pseudo) executable sequences.

In a first phase the generator translates the xADL description to this internal description which includes explicit operations to cover all architectural properties to be simulated. The actual generation of the two simulator parts (interpreter and binary translator) is carried out in the second phase where each element of the internal description is generically translated to the desired form of code. Because basic elements of the internal description have simple and well-defined semantics this translation to C code can be performed without any special cases and independent from other elements.

Besides a rough overview, Figure 3.1 illustrates some details of the internal description: Sequences and memory-elements which are described in the following sections.



```
<adl:Architecture
 <ABI name=mips-elf
  ...
 </ABI>
 <Configuration ...
 <BinaryFormat ...
 <RegisterType ...
 <UnitType ...

 <Unit name="FE">
  ...
 </adl>
```

```
void inst_35_stage3(...)
{
 fixint<16> DCa_writeH;
 fixint<32> ME_ST_Rd_i;
 fixint<32> ME_ST_Rt_i;

 ME_ST_Rt_i = EX_Rt_o;
 ME_ST_Rd_i = EX_Rd_o;
 ME_ST_A_o = ME_ST_RD_i;
 ...
 ...
}
```

Figure 3.1: Overview of xADL tools and the Simulator Generator

3.4.1 Memory-Elements

The internal description can be seen as virtual programming language: calls (that are closely related to those in the xADL description) form instructions of this language. Variables for this language are called *memory-elements* or *mem_elements*.

Each mem_element has a type and some parameters that specify its properties. One important property is the bitwidth of the memory-elements which may be an arbitrary positive integer. Memory-elements may be constant and have different visibility scopes: Basically we distinguish between global and local mem_elements but for the latter ones two versions exist.

Memory-elements with global scope have a unique identifier and are visible within all pipestage-functions of all instructions. Global mem_elements are eventually translated to global C variables. They are used to represent the architectural state (mainly registers and memories) of a described architecture.

The first kind of local scope is called temporary, is local for a single pipestage-function and is translated to a local variable for the interpreter. The second kind is operand scope which is used for instruction operands that can be immediate values or register indices and are visible (and have unique identifier) within an instruction.

Besides "normal" memory-elements that store a single value with the specified bitwidth there are two types of array memory-elements: registers and memories. For both types an additional parameter specifies the size of the array in registers or bytes (for memories). Instead of using array elements directly (as arguments or results) in the call sequences, special array access mem_elements are needed. These can be used like any other memory-element and encapsulate the array and an index, itself a memory-element, that specifies the position to be used inside the array.

3.4.2 Call Sequence

Call sequences are a simple sequential succession of calls (basic operations) and condition elements. Calls have a number of mem_elements as arguments and results and perform one of a few predefined logical and arithmetic operations on them. The semantics of these operations is identical to those of the xADL built-in calls, because most of them are translated one-to-one. Other than in the xADL description, there is no *cmove* (conditional move) call in the sequences of the internal description, because control flow is modelled explicitly in form of condition elements.

The structure of condition elements is very simple: they consist of a condition and two call sequences (if_sequence and else_sequence). At runtime one of these sequences is executed depending on the value of the condition. With these simple control flow elements it is not possible to represent an arbitrary control flow graph (CFG) but only simple or nested if-then-else structures. For the translation of operations containing *cmove* calls and other simple conditions this strictly hierarchical representation is sufficient and enables easy translation to C-syntax.

3.4.3 Condition Tree

Conditions in condition elements always compare memory-elements (that must have matching bitwidth parameters) pairwise with each other. Possible relational operators that may be used for their comparison are:

	Semantics	Operation		Semantics	Operation
eq	equal	$(A = B)$	**lt**	less than	unsigned $(A < B)$
neq	not equal	$(A \neq B)$	**ge**	greater or equal	unsigned $(A \geq B)$
gt	greater than	unsigned $(A > B)$	**le**	less or equal	unsigned $(A \leq B)$

Table 3.1: Possible Condition Operators

It is possible to build more complicated conditions by combining simple comparisons with logical AND and OR operations. Together with constant memory-elements the resulting condition tree has the expressiveness to represent all needed (unsigned) conditions.

3.4.4 Building up Internal Description

As stated before the call sequences are built according to the operations along the instruction path of the xADL description. Many of these operations (more precisely their calls) are directly translated in sequential order. For operations that include *cmove* calls or additional architectural semantics that is not directly represented in the instruction path (e.g. register write priorities) a more complicated translation is needed that might involve condition elements and non sequential description construction. While the description of a specific instruction is generated an insertion pointer defines the position inside the call sequence, where the next call is to be inserted. For direct sequential translations this pointer is just directed to the next position. For conditional instructions (e.g. "branch not equal“ on MIPS R2000) parts of the operations have to generate their calls inside the if_sequence of a condition element - this is achieved (transparently) by setting the insertion pointer to that sequence.

In more complicated situations, where mem_elements are affected by multiple *cmove* calls, condition elements can be transparently combined with logical AND or OR and form new condition elements.

Chapter 4

Dynamic Translator

In the previous chapter concepts of our simulator generator and the structure of the internal description, used to express instructions, were described. Details of the transformations from xADL description to this internal instruction based description are not described in this work but can be found in [Fellnhofer, 2008]. Instead it is assumed that an internal description is available for each instruction of the architecture, including forwarding mechanisms, handling of register priorities and all other needed architecture dependent constructs.

The focus of this work is on how such internal descriptions can be used to generate a dynamic translating simulator, which additional information is needed and how it can be extracted out of xADL descriptions.

4.1 Related Work

4.1.1 History of JIT

In [Aycock, 2003] the history of just in time (JIT) compilation is summarised. Starting at 1960 many different forms of JIT systems are presented and their evolution is observed. Different types of simulators are categorised in four classes:

- **first generation:** classical interpreters that process one instruction after the other
- **second generation:** simulators that translate single instructions to host code
- **third generation:** translators operating on static sequences of instructions, mostly basic blocks

- **fourth generation:** simulators having dynamic selected translation units (traces)

Fundamental parts of common fourth generation simulators are outlined: different concepts of profiled execution and hot path detection are briefly described.

4.1.2 Shade

Shade is a fast instruction set simulator presented in [Cmelik and Keppel, 1994]. It uses dynamic translation on basic block level and supports profiling of simulated applications. As basic blocks are usually rather small, two adjacent blocks are chained to each other. This means that the controlflow instruction at the end of a translated block that normally jumps to the main loop, is patched: controlflow is then directly transferred between two translations without any overhead in the main loop. Ways to cache and find translations are presented along with general considerations about compiletime/runtime tradeoffs.

4.1.3 Dynamo

In [Bala et al., 2000] a dynamic optimisation system is presented: Binary code is transparently profiled, optimised and recompiled during its execution. Several basic blocks of hot code form a so-called *fragment* and are compiled into a translation cache. A lightweight compiler speculatively reorders basic blocks inside fragments and removes redundancies. In contrast to static profile techniques, dynamic profiling of execution allows to find dynamically changing working sets, which can be recompiled at runtime to obtain high locality. A heuristic cache management flushes the translation cache whenever the rate of compiled fragments increases, which may indicate a new working set in the executed program. The proposed approach is able to dynamically optimise the interface between dynamically linked libraries and even increases the execution speed of many statically optimised applications.

4.1.4 PIN

The work of [Luk et al., 2005] presents the **PIN** instrumentation framework that uses dynamic binary compilation (without translating between different instruction sets) to achieve high performance code analysis at runtime. Basic blocks of binary applications are enriched by instrumentation instructions and optimised at runtime. Big parts of the framework are architecture independent, which enables at least simple instrumentation analysis, like trace analysers or memory bug checkers, to be generic and usable for all supported architectures. Just in time optimisations include register reallocation, register liveness analysis and instruction scheduling. Traces build the compilation unit - they are linked to each other even on indirect jumps, using software branch prediction. The average performance overhead of the framework (without active instrumentation) is in the range of only 5 to 25 percent for the preferred architectures: X86 32-bit, X86 64-bit and Itanium.

4.1.5 Strata

The retargetable translation framework **strata** is described in [Scott et al., 2003]. Its concept is based around a virtual machine which abstracts the host architecture. Binary translation is done at the basic block level. Common services can be shared between different architectures, while only some reconfigurations and moderate implementation effort is needed to support new architectures. Two example implementations for different architectures are presented: A call monitor used to execute untrusted binaries that intercepts iniquitous system calls and an environment for fast prototyping of simulators.

4.1.6 UQDBT

In [Ung and Cifuentes, 2000] the **UQDBT** framework for dynamic binary translation is presented. The possibility to configure this framework for different source and host architectures, and its clear separation of architecture independent analysis from machine specific concerns are highlighted. An abstract intermediate representation is used as base for analysis and optimisations. In [Ung and Cifuentes, 2001] a method (using edge weight count between basic blocks) for finding hot paths of simulated applications is demonstrated. Combination of basic blocks to larger translation units and their reordering in memory is used to improve performance. Frequently executed code parts are additionally optimised as this overhead is justified by better runtime performance.

4.1.7 bintrans

A machine-adaptable dynamic binary translator that does not use an intermediate representation is **bintrans** [Probst, 2002]. Two machine descriptions (for source and host architecture) are used to generate a simulator that emulates a Linux system. A matching-algorithm is used to find an appropriate host-instruction sequence for each source instruction. Key issues for fast simulators generated by this approach are efficient mappings of condition codes (flags) and the elimination of dead condition codes (simple liveness analysis). When the host architecture has enough registers a static mapping of source registers is used to increase the efficiency of translated code. Furthermore this paper explains the problem of self modifying code and presents appropriate counteractions for different architectures.

4.1.8 Hot Code Recognition

The framework described in [Shi et al., 2007] focuses on different types of hot code recognition. Similar to the above described approaches the framework translates source instructions to an intermediate representation which is eventually (after some transformations) compiled to executable host instructions. Block based analysis of hot code is compared to edge based methods - general considerations for tradeoffs between analysis cost and analysis effectiveness are made. A new form of profiling is described that uses binary codes to identify the hottest paths of an application - this information is used by a translation cache manager to privilege hot translations.

4.1.9 High Speed CPU Simulation using JIT Binary Translation

The paper of [Topham and Jones, 2007] presents a retargetable full system simulator. Its goal is to achieve high simulation performance for embedded processors (with memory management units) in a state and cycle accurate simulation model. Dynamic binary translation is used in combination with interpretation techniques. The translation of basic blocks is performed in so-called epochs: after a certain number of blocks have been interpreted (and recorded to a so-called Epoch Block Cache) a heuristic determines hot blocks for compilation - the next epoch begins when compilation is finished. This ensures that compilation is only performed when enough benefit is expected - because after compilations a complex reorganisation translation caches is required. The simulator is able to simulate the bootprocess of a Linux operating system at a rate of about 150 instructions per second (on a 3GHz 4-processor server).

4.1.10 iboy

The handcrafted full system emulator **iboy** uses dynamic binary translation to speed up the simulation of a Z80 processor on the ARM architecture [Fellnhofer and Rigler, 2006]. The very lightweight compiler is fast enough to compile all executeable code at runtime; an additional interpreter is not needed. Templates specify a series of host instructions for each instruction of the source architecture (Z80). These templates are specialised for constant flags (condition codes). A simple analysis avoids the computation of dead flags. Interrupts of the simulated system are only processed at the end of basic blocks which increases simulation speed at the cost of imprecise timings.

4.2 LLVM

LLVM is a compiler framework developed under an open source licence. It uses a special architecture independent bitcode (or intermediate representation, short IR) internally for the representation of code. It operates on a low level virtual machine (hence the name LLVM), is strictly typesave and can be stored in three different forms: human-readable assembler-like syntax, compact bitcode for storage on disk and an easy to manipulate tree form to reside in memory. All these forms are equivalent and the LLVM framework can translate between them. [Lattner and Adve, 2004]

LLVM includes many analysis and optimisation passes that can be applied to the intermediate representation - making these passes architecture independent. For the backend (code generation) however architecture specific passes are needed. LLVM includes ready-to-use backends including X86 (32 and 64 bit) and PowerPC and is extensible for new architectures. Frontends that translate different programming languages to LLVM bitcode complete the framework to a full blown compiler. The most interesting part of LLVM (regarding this work) is however the JIT framework with its capability to translate LLVM bitcode at runtime and directly execute it afterwards.

Besides frontends that translate sourcecodes of different programming languages to IR, LLVM provides a C++ API to build up IR via library calls (in tree form). The C++ convenience class *IRBuilder* further simplifies construction of LLVM intermediate representation using this API.

The relevant LLVM JIT libraries are linked to the simulator framework and thus can use *IRBuilder* for the dynamic translator to generate code. A class *llvmc* derived from *IRBuilder* functions as central point of dynamic code generation.

4.3 Generation of LLVM Code

As mentioned in Section 3.4.2 the internal description consists of a series of micro operations (calls) in a simple if-then-else structure, operating on memory-elements. For the binary translating part of the simulator each of these micro operations is translated to an invocation of a C++ wrapper function implemented in our *llvmc* class. For each instruction and pipeline stage a separate pipestage-function consisting of the described wrapper invocations is generated (see example Listing 4.1). Memory-elements are translated to objects of special *llvm_sim_type* classes (see below).

However the wrapper functions do not directly execute the operations specified in the internal description but generate code that does this. In an analogues manner *llvm_sim_type* classes do not store values inside their data members but take care of memory space in the generated code.

Listing 4.1: A simplified pipestage-function

```
void llvm_instr_69_stage1(union_ops *ops)
{
  llvm_tmp llvm_DE_ImmW_i(16, &llvmc); /* local variable */

  ...

  wrp_move(llvm_DE_ImmW_i, ops->ops_69.ImmW);/*move*/
  wrp_sext(llvm_DE_ImmWs_o, llvm_DE_ImmW_i); /*signextend*/

  ...
}
```

4.3.1 Memory-Elements

Memory-elements are differently prepared for dynamic translation depending on their type. For each memory-element an object of the *llvm_sim_type* class or one of its subclasses is instantiated. These classes primary provide methods to load and store values, specify the bitwidth of the corresponding *mem_element* and are used as arguments to the wrapper functions. For global mem_elements, global C variables of type *llvm_glb* are generated whereas local mem_elements are generated as local variables of type *llvm_tmp* inside the pipestage-functions.

At the instantiation of a *llvm_glb* object a pointer defines to which C-variable it corresponds. This enables the generated code to directly access global variables used by the interpreter part of the simulator and makes it possible to use them in parallel without additional memory transfers. When executing the *store()* method of a *llvm_glb*, code is generated that stores a value (provided as argument) to the location of the C-variable; the *load()* method generates code that reads the location and returns its value. Note: A value in this context must be seen as a placeholder for a concrete value like a temporary variable and is represented by the LLVM *Value* class.

For local mem_elements another semantics is implemented: At the instantiation of the *llvm_tmp* object (inside the pipestage-function) code is generated that allocates local memory using *alloca()*. This allocated memory is then used by all subsequent *load()* and *store()* invocations.

Special mem_elements used to represent registerfiles and memories are handled in separate sub classes, which provide an additional method to specify an index or address in form of another *llvm_sim_type* object.

Operands of instructions are provided as arguments for pipeline stage functions and are treated as constants in the generated code. Like for constant mem_elements their value

Listing 4.2: A simple wrapper function

```
void wrp_move(R0 &res0 , A0 &arg0)
{
  res0.store( arg0.load() );    // res0 = arg0;
}
```

gets hardcoded in the dynamic compiled code.

The simplest wrapper function in the llvmc class is *wrp_move* and is shown in Listing 4.2. It has only two arguments and generates code that transfers code from one to the other using their *load()* and *store()* methods.

4.3.2 Conditional Structures

As already mentioned the *llvmc* object encapsulates the code generation of the binary translator. When wrapper functions are called one after the other, code is continually appended to the current chunk of compiled code. More precisely such a chunk is a LLVM *Function* object that may contain several *BasicBlock* objects representing basic blocks of the host architecture. Wrapper functions for built-in calls always insert code to the last basic block in the current function. Additional basic blocks are only inserted trough transformations of conditional structures. For each if-then-else structure three additional basic blocks and four (conditional) branches are generated (see Figure 4.1).

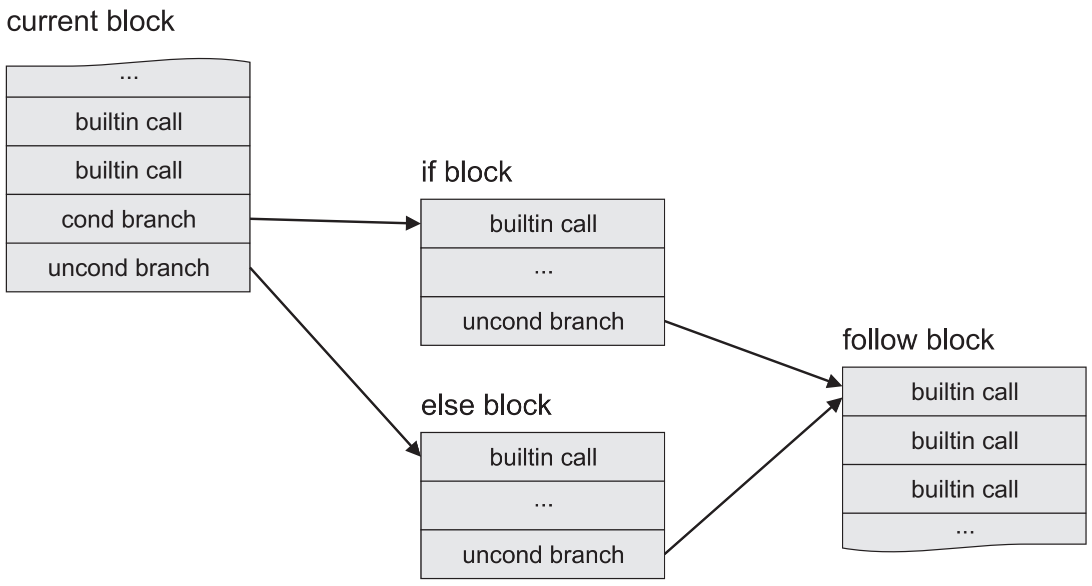


Figure 4.1: Condition nodes are translated to a series of host basic blocks.

4.4 Basic Blocks

Before the first wrapper function generates its code, the *next_function()* method of the global *llvmc* object has to be called. It creates a new function with its first basic block. Until the invocation of *finish_function()* closes the current function, all called wrapper functions emit their code to this function - even if these calls originate from different pipestage-functions. Therefore it is possible to compile all pipestage-functions that are needed to perform the simulation of a full clock cycle into a singe function.

For instruction set based translators the smallest unit of translation is one instruction, whereas our concept is based on direct pipeline simulation where instructions are executed interlocked and the clock cycle is the granularity where compiled blocks are cut.

In theory a simple translating simulator could compile each clock cycle to a separate binary block, execute it, update the pipeline and continue with the next cycle. This however would lead to very bad performance as the compiler would not have enough "sight" to make useful optimisations. Therefore as much as possible has to be compiled into one binary block to obtain a faster simulator.

As indicated in Section 3.1.4 for each simulated clock cycle one pipestage-function per instruction in the pipeline is compiled. Additionally the corresponding epilogue pipestage-functions (that have identical layout) are compiled in a second pass. For bigger binary blocks that require more than one clock cycle compiled into a LLVM function, additional code is generated to increase the global cycle counter between clock cycles.

To determine the next instruction to be compiled, an instruction decoder needs the simulated program counter (PC). As decoding has to be done at compiletime the sequence of corresponding PCs for the compiled chunk has to be known a priory.

Jump instructions that (conditionally) modify the programflow restrict the size of compiled blocks, as the PC value (after their execution) cannot be unambiguously determined at compiletime. The idea is to compile a sequence of source instructions until the occurrence of the first jump instruction in conjunction (which corresponds to a source basic block) and start the next binary block afterwards. In xADL jump instructions are not explicitly specified, instead an analysis pass is needed to identify them by their behaviour (see Section 4.7.3).

When the code for a complete block is generated, the resulting LLVM *Function* can be compiled. This is accomplished trough its *getPointerToFunction()* method, which transparently compiles the function in memory and returns a C-Function pointer to it, which can be used to execute the compiled code.

4.4.1 Overlapping Instructions

The advantage of a dynamic translator over a simpler interpreter is its higher execution speed of the simulated program. The compilation of one block is however much more time-consuming than its interpretation. Therefore it is inevitable to store compiled blocks in some sort of cache and reuse them as often as possible: Whenever the simulated program counter points to a basic block that is already in the cache, it can be executed without being recompiled (see Section 4.5).

When looking into details of pipeline simulation the following fact has to be considered: a jump instruction (conditionally) alters the simulated PC in one definite stage in the pipeline. After this stage has been executed the PC points to the beginning of the next basic block, however there are still instructions in the pipeline belonging to the current basic block, because jump instructions are executed in parallel with other instructions. In other words, compiled blocks can only be cut at clock cycles and instructions sprawl over several clock cycles which implies that at least some instructions are cut and their pipestage-functions are distributed over at least two compiled blocks.

Therefore when the next basic block is executed, parts of instructions from the previous block have to be finished (their remaining pipestage-functions have to be executed/compiled; see Figure 4.2).

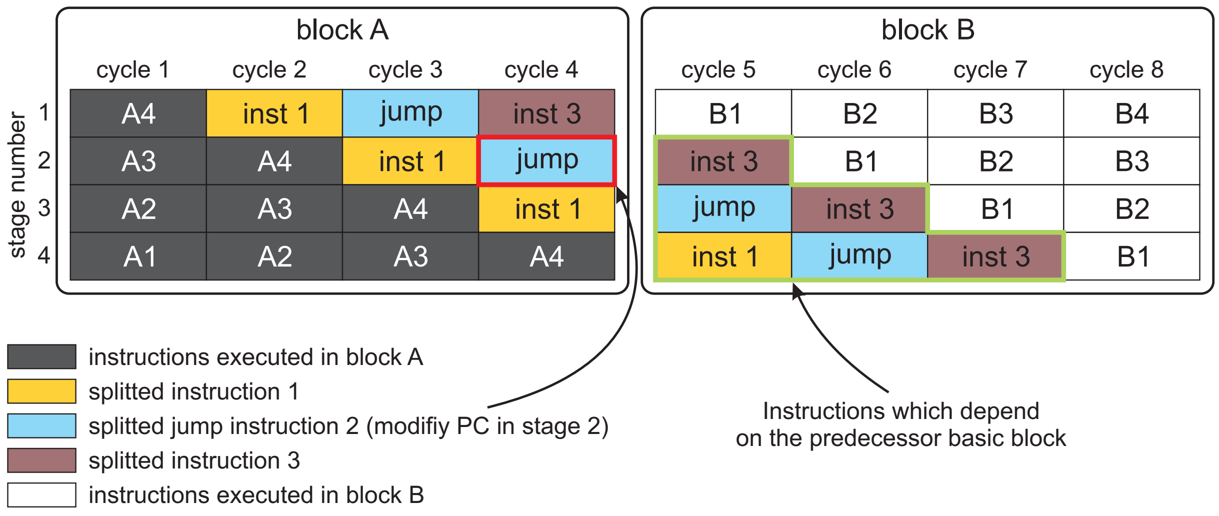


Figure 4.2: Parts of block B depend on its predecessor block A.

The problem is that a single basic block can have different predecessor basic blocks that may need different instructions to be finished. When the whole behaviour of a block is compiled and cached, the block might be improper if it is used in a context where a different predecessor block had been executed.

To overcome the problem it is possible to either compile in dynamic code that interprets these remaining parts of the different predecessor basic blocks or to compile and store multiple versions (one for each sort of predecessor) of basic blocks in the cache. Although the latter needs larger caches and more compilation time we preferred it because of its assumed better runtime performance.

Several predecessor blocks that end with the same instruction sequence do not require additional versions of blocks. This is the case when one basic block is a subset of an other basic block, which is common in situations as illustrated in Figure 4.3. Therefore it is possible that one compiled block is the successor of several (other) blocks.

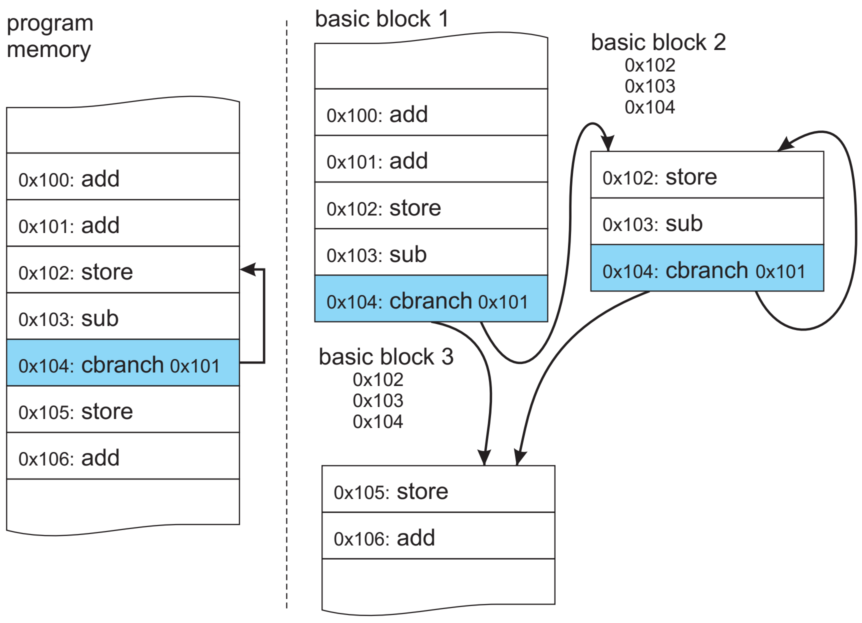


Figure 4.3: One compiled block may have several (similar) predecessor block.

Two compiled blocks that share a common successor block must have the same signature - this means that their terminal instructions have to be identical. The number of relevant instructions is determined by the size of the processors pipeline.

Figure 4.4 shows the same code fragment as 4.3 but translated for an architecture with more pipeline stages - the small numbers above basic blocks illustrate the signature of allowed predecessor blocks.

4.5 Caching

As already mentioned, caching of compiled blocks is an essential functionality of each fast dynamic binary translator. Such caching mechanisms usually consist of two parts: a (dynamically growing) portion of memory to store compiled blocks and some form of

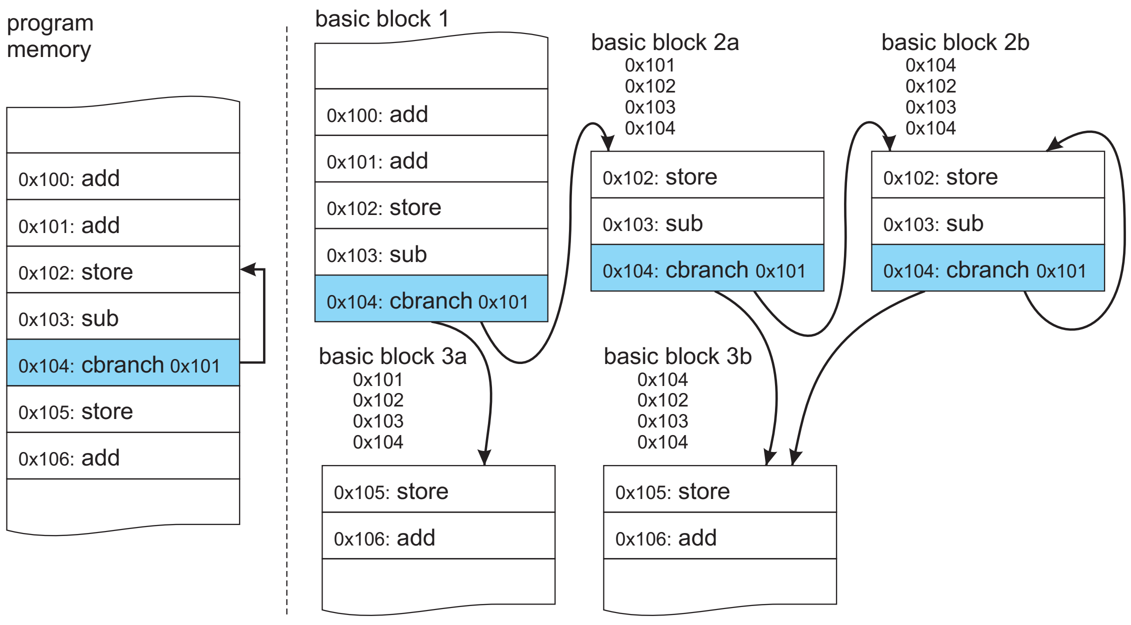


Figure 4.4: A longer pipeline requires multiple versions of blocks.

index that maps identifications (keys) of basic blocks to locations inside this memory.

For translators that operate on the abstraction level of the instruction set it is often sufficient to use the PC of the first instruction in a basic block as its identification. Our translator, that simulates the internal structure of the pipeline, employs a more complicated approach: the identification of a compiled block is composed of the addresses of all instructions that are in the pipeline buffer in the first cycle of the block. To obtain these addresses the value of the program counter is stored in the pipeline structure whenever a new instruction is decoded (see pipeline data structure in Section 3.1.3). Depending on the predecessor block this approach leads to several different keys for one source basic block - and thus multiple versions of the same block are distinguishable from each other and can be stored in the cache.

The JIT framework of LLVM takes care of allocating memory to hold compiled binary executable code. When the intermediate representation of a block is compiled trough the *getPointerToFunction()* method, LLVM is in control of the associated allocated memory, and provides mechanisms to access it from outside: LLVM *Function* objects can have a name that identifies them. This name (a string) can be used to find the location of a distinct function. Because of the poor performance this mechanism can hardly be used to check for already compiled blocks (using keys encoded as strings).

Instead a separate hashmap is used to find compiled functions in this memory area. The key to this hashmap consists of integers (addresses of source instructions, as described above) which are easier and faster to process than strings. For each version of a source basic block a small data structure is added to this hashtable. It basically stores pointers to the LLVM *Function* object (in intermediate representation form) and to the location of the executable compiled block. Additional data elements are used for optimisations (see Section 4.7) and to keep track of some statistics like the size of basic blocks or the number of versions compiled for each source basic block.

When source applications are simulated that modify or create program code (self modifying code) reasonable precaution has to be taken to update the translation cache - such precautions are missing in the current implementation.

4.6 Compilation in the Framework

As described in Section 3.1.4 the simulators mainloop simulates one clock cycle of the source architecture in each iteration. For the interpreting simulation this comprises decoding of the next instruction, updating the pipeline buffer, executing the pipestage-functions and epilogues in the described order and increasing the global cycle counter.

4.6.1 Compilation of Basic Blocks

In the dynamic binary translator the current state of the pipeline buffer is evaluated (prior to the execution of pipestage-functions) to check whether the current cycle can be compiled - in the current implementation system calls can only be executed in the interpreter, hence not all cycles are allowed to be compiled. When a cycle is compileable the key of the current pipeline state is calculated. This key corresponds to the basic block starting at the cycle that will be executed in the current iteration of the main loop.

Before a new LLVM *Function* for this basic block is instantiated a simple logic estimates whether it is worth to compile this basic block. For this purpose the cache provides a special method called *is_often_used()* that counts the occurrences of each key and returns *true* only if the specified key has been used more often than some predefined threshold value. This shall assure that only frequently executed blocks are compiled. Compilation of rarely executed blocks would degrade overall performance as the compilation overhead cannot be compensated through faster execution times in this case.

When the threshold value is reached for a given key, a new *Function* object is instantiated in the *next_function()* method of our *llvmc* object. Before the first interpreter pipestage-function (last stage of last instruction in the pipeline buffer) is executed, the corresponding compiler function is called and generates its code into the newly created function. All subsequent stage and epilogue functions of this cycle are alternately compiled and interpreted in the same way. This means that compilation is done in parallel to the interpretation which has some advantage regarding compiletime constant values, which are described in Section 4.9.2.

All following cycles are handled similarly, except that no key is computed and no new function is created. Instead the generated code is appended to the current function until the end of the basic block is reached (a cycle is conditionally modifying the program counter). Between two cycles, code is inserted to increase the global cycle counter. In the end *finish_function()* is called to compile the LLVM *Function* object and save the executable output in the cache.

4.6.2 Pipeline State Restoration

The cache is queried after the key of a compileable cycle is computed. When the associated compiled binary block is found in the cache, it is executed - omitting the more time-consuming interpretation. Thereby the compiled block alters the architectural state as it would have been done by the interpreter. However a compiled block does not update the pipeline buffer nor does it decode instructions while it is executed. Because a compiled block may consist of more than one cycle (and each cycle shifts the pipeline by on step), the pipeline buffer may be in an outdated state afterwards. To continue simulation, the pipeline buffer has to be restored to the state that it would have had past the interpretation of the same block.

As this state contains only information about the (static) program code and the history of the program counter, it is mostly independent of the architectural state and therefore identical after each execution of the same binary block - except for the outcome of the terminating jump instruction.

Thus the state after the teethed interpretation/compilation run of a basic block can be used as base for the state restoration: At compiletime, after code generation for the last cycle has been done, the state of the pipeline buffer can be stored in the cache, side by side with the binary block. To save space only the program counter sequence of the pipeline buffer is stored, omitting actual instructions and instruction operands of this structure. When the pipeline has to be restored, the decoder can use this sequence to reinitialise most of the pipeline buffer. Only the latest instruction in the pipeline is determined by the PC in the architectural state which depends on the outcome of the jump instruction that ended the basic block - this information is used to complete the state restoration.

To reduce the overhead of pipeline restoration, the stored restoration PC sequence (and the PC in the architectural state) can be used to directly obtain the key of the binary block that has to be executed next. That way the costly pipeline restore can be omitted, if the required successor basic block is already compiled in the cache.

4.7 Linking of Basic Blocks

The dynamic translator described so far is able to run most source applications faster than the interpreter that we generated based on the same description. A performance profiling using gprof indicates that the cache queries require a big portion of the overall simulation time. This portion can be minimised by a huge reduction of cache queries, which is possible by the following considerations:

4.7.1 Uncertainty of Successor Blocks

As described above the only uncertainty about the successor of a binary block is the value of the PC after the terminating stage of the jump instruction has been executed. Depending on the nature of a jump instruction this uncertainty is restricted to some extent.

Different types of jumps are described in Section 1.1.2 - their most important property regarding dynamic translation is the amount of different possible jump targets.

Simple unconditional jumps to a single fixed address are easy to handle in dynamic binary translators: as their destination is known at compiletime they even do not have to end a basic block in many cases. The same is true for unconditional jumps that have a fixed PC relative offset as the initial PC can be interpreted as a constant from the compilers point of view.

The other extreme are jump instructions with a virtually unlimited number of possible jump targets. Such computed jumps either take the value of a register as target (e.g. return from function call) or arithmetically compute the value of the target location (e.g. jump in a switch statement).

The most common jump instructions in many architectures (for most applications) are however conditional jumps that have a fixed target or a constant PC relative offset. For binary blocks that are terminated by such jumps there are exactly two possible successor blocks, one for "jump taken" and one for "jump not taken".

4.7.2 Return Values of Basic Blocks

The idea of basic block linking is to omit cache lookups for jump instructions where only a limited number of successors exist - this is done by storing successor pointers inside the basic block structure in the cache.

The basis of our block-link implementation is that each compiled block returns an ID that indicates whether a jump was taken (and which type of jump it was). When a new LLVM

Listing 4.3: Pseudocode of LLVM IR that shows how return_value is assigned when two jumps are possible in the last cycle of a block.

```
one_source_basic_block() {
  return_value = 0;       // ID for 'jump not taken'
  ...
  if ( exception ) {
    PC = 0x400000;        // exception address
    return_value = 1;     // ID for 'exception jump'
  } else if ( jump_condition ) {
    PC = PC - 16;         // constant offset
    return_value = 2;     // ID for 'jump taken'
  }
  ...
  return return_value;
}
```

function is created, the *next_function()* method generates a local variable *return_value* that is initialised by zero and *finish_function()* creates an appropriate instruction to return this value.

Whenever an instruction modifies the program counter, *return_value* is set to a certain value: a positive integer value for modifications that have a priori known targets and -1 for unknown jump targets.

Note: The outcome of a PC modification in a conditional jump may very well be known a priory although it can not be predicted whether the modification is done at all.

In theory the last cycle in a compiled block can modify the program counter at different points - for example when a stage, that can throw an exception is executed in parallel with the determining stage of a jump instruction. In this case the a priori known modifications are numbered serially starting by 1. Listing 4.3 shows example pseudo code of generated LLVM IR that determines the *return_value* depending on the taken jump target.

The described semantics of the return IDs assures that whenever a compiled block is executed and returns the same value (except for -1) for several times, the same jump target has been taken and the same successor block has to be executed next. Thus to avoid most of the cache lookups the following logic has been implemented: Whenever a binary block is executed and returns a value ≥ 0 the cache is queried for its successor block. When this lookup is successful, a pointer to the successor block can be stored inside the small data structure of the originating basic block. For this purpose the data structure is extended by a successor array that stores these pointers using the returned ID as index.

For subsequent executions of the same block, the (non negative) return value is used to index the successor array. If that entry is existent (pointer is not null) it must point to the proper successor block, which then can be executed without any computations of keys or cache lookups.

The benefit of block linking is that after some simulation time all needed blocks for a frequently executed (inner) loop of an application are likely to be compiled and linked to each other - in this case their simulation can sometimes be done without any cache lookups or pipeline buffer restores.

4.7.3 Jump Analysis

Jump instructions in the xADL language are not explicitly defined as such. Therefore an analysis is required to identify instructions that modify the program counter and characterise them into one of the above categories to make linking of blocks possible. Basically this analysis is done in the following way: whenever a built-in call of an instruction uses the program counter as its result the path of the corresponding value is retraced to its origin. In other words all built-in calls and components involved in the computation of that value are checked.

If only simple built-in calls (like *add*, *sub* or *shift*), constants, immediates and the (previous) value of the PC are involved the jump is marked as "known" because its target can be determined at compiletime. Otherwise the jump is marked as "unknown" e.g. when register values are used for its computation.

In xADL it is possible to explicitly describe the incrementing of the program counter for all (even non jump) instructions. These modifications are ignored by the analysis (based on their simple form of adding a constant offset to the program counter).

For each instruction the analysis generates a jump mask: a bitpattern that contains one bit for each stage of the instruction that indicates whether the stage is able to change the programflow. This mask can efficiently be used in the simulators mainloop to check if the end of a basic block has been reached.

For the linking of blocks, the internal description of each PC modification is enriched by a special marker call which is converted to a method call of the global llvmc object for the dynamic translator (see Listing 4.4). Two different versions of these markers are possible: *unknown_jump_taken()* and *known_jump_taken()*. Their invocations generate code that provides the above described return values.

Listing 4.4: A simplified pipestage-function of a conditional jump instruction

```
void llvm_instr_50_stage2(union_ops *ops)
{
  ...

  /* condition of the jump/branch instruction */
  llvmc.CreateCondBr( /*[CONDITION]*/ ), IfBB, FollowBB);
  llvmc.SetInsertPoint(IfBB);
    wrp_move(llvm_pc, llvm_EX__pc); /* modify the PC */
    llvmc.known_jump_taken();        /* mark modification */
    llvmc.CreateBr(FollowBB);
  llvmc.SetInsertPoint(FollowBB);

  ...
}
```

4.8 Traces

Linking of binary blocks reduces the time spent in the framework of the simulator to a minimum. In the ideal case all frequently executed blocks are compiled and linked to each other - only for computed jumps and the execution of system calls some overhead is needed. Most of the overall runtime is spent for the execution of compiled blocks then. To further increase simulation speed in this scenario the runtime efficiency of binary blocks has to be improved.

For example this can be achieved by avoiding unnecessary computations and memory transfers, or by omitting conditional operations when their condition cannot be fulfilled. These kind of optimisations can be done by the LLVM library on the intermediate representation of a source architectures basic block - the compilation unit of our so far described translator. LLVM optimisation passes are briefly described is Section 4.9.1. For them to be effective the size of the compilation unit has to be large enough because no assumptions about the architectural state outside its borders can be made. Besides higher optimisation opportunities, larger compilation units result in code that has higher locality and can therefore (potentially) be executed faster.

For our translator (depending on the architecture) a compiled block typically consists of just a few clock cycles and evidently cannot contain any loops. This induces a severe restriction to the optimisation potential, especially when considering that it is very likely that most of the simulation time is spent inside (small) core loops of the source application.

The solution for better optimised compiled blocks is the creation of larger compilation units: traces. These traces however are not used instead of basic blocks, but enrich the translator with the possibility to better optimise frequently executed parts of an application (hot code). This means that compilation is done incremental in two grades: first basic blocks and then traces.

To identify hot code several different approaches are proposed in literature (see Section 4.1). Many of them are either based on counting executions of source basic blocks or on counting most frequently taken paths between them.

We count execution frequencies of compiled blocks that correspond to special versions of source basic blocks. Based on these frequencies, blocks are selected for compilation of a trace. Different predecessors lead to new versions of basic blocks and eventually result in traces that have a composition that is similar to traces created by analysing path frequencies.

4.8.1 Recompilation of Hot Paths

As previously mentioned the cache holds a data structure for each compiled block. It contains a pointer to the LLVM intermediate representation, which is kept even when the block has already been compiled.

When a compiled basic block is executed more often than a certain number, the compilation of a new trace begins: a new LLVM *Function* object, that represents the trace, is created. The first instruction in this function is a function call to the intermediate representation (itself a *Function*) of the block that initiated the trace compilation. Then code is appended that evaluates the return value of this function call. As this value corresponds to the ID of the successor block, the successor array, initially used to link basic blocks, can be used to determine which block has to be called/compiled-in for a certain return value.

For each known ID in the array a separate case for the return value is generated that contains a call to the *Function* of the appropriate successor block - for IDs that are not stored in the successor array a return instruction (that terminates the trace) is generated. For function calls of successor blocks in turn similar evaluations of their return values are generated in a recursive way until no more known successors are found (or a certain number of blocks is exceeded).

So the trace compilation gets triggered by a block executed more often than some trace threshold value but includes all reachable (by uncomputed jump) blocks that were executed more often than the basic block compile threshold.

Of course the graph built up by connections between basic blocks in a trace may contain loops. Therefore (to avoid infinite large traces) a trace may not contain two calls to the same function. Instead all calls to a function that has already been called are replaced by a branch to the first calling cite of the function. These IR branch instructions form loops in the compiled trace.

Before a newly generated trace is compiled all function calls are inlined to obtain a big function that does not call any subfunctions and can be optimised as a whole.

When a trace is compiled, its executable form is stored in the cache and replaces the executable form of the basic block that triggered the trace compilation - the key for this basic block then points to the head of the trace.

4.8.2 Return Values of Traces

After a trace has be executed, one of its successor blocks might demand a run of the interpreter (and compiler). But this requires the pipeline buffer to be restored (as described in Section 4.7.2 after the execution of basic blocks). The pipeline state after the run of a trace however depends on its last executed basic block. Therefore return values of traces are composed of two components: the jump ID and a "block number" that identifies the last executed block.

When a trace is compiled, a list of all contained basic blocks is added to its data structure in the cache. The block number, encoded in the return value, indexes this list and leads to the data structure of the last executed basic block, which contains the information required for pipeline restoration. Additionally the successor array which is indexed by the jump ID can be used to link adjacent blocks (or even traces) that have not been compiled at the creation time of the trace.

Figure 4.5 shows an example trace with four possible exit points (one for each contained basic block). White entries in the successor array represent links to other blocks inside the same trace.

4.9 Optimisations

The compilation of traces increases the size of compiled blocks which leads to high optimisation potential as many dependencies of memory-elements can be resolved within the compilation unit. LLVM provides a number of optimisation passes that exhaust much of this potential. For some kinds of optimisations however no appropriate passes exist. Some others would require additional information about semantics of memory-elements from outside the compilation unit that cannot be described in the LLVM intermediate representation. Therefore before optimisation passes are run, the IR is enriched by some operations that account for those deficiencies and eventually lead to better optimised code.

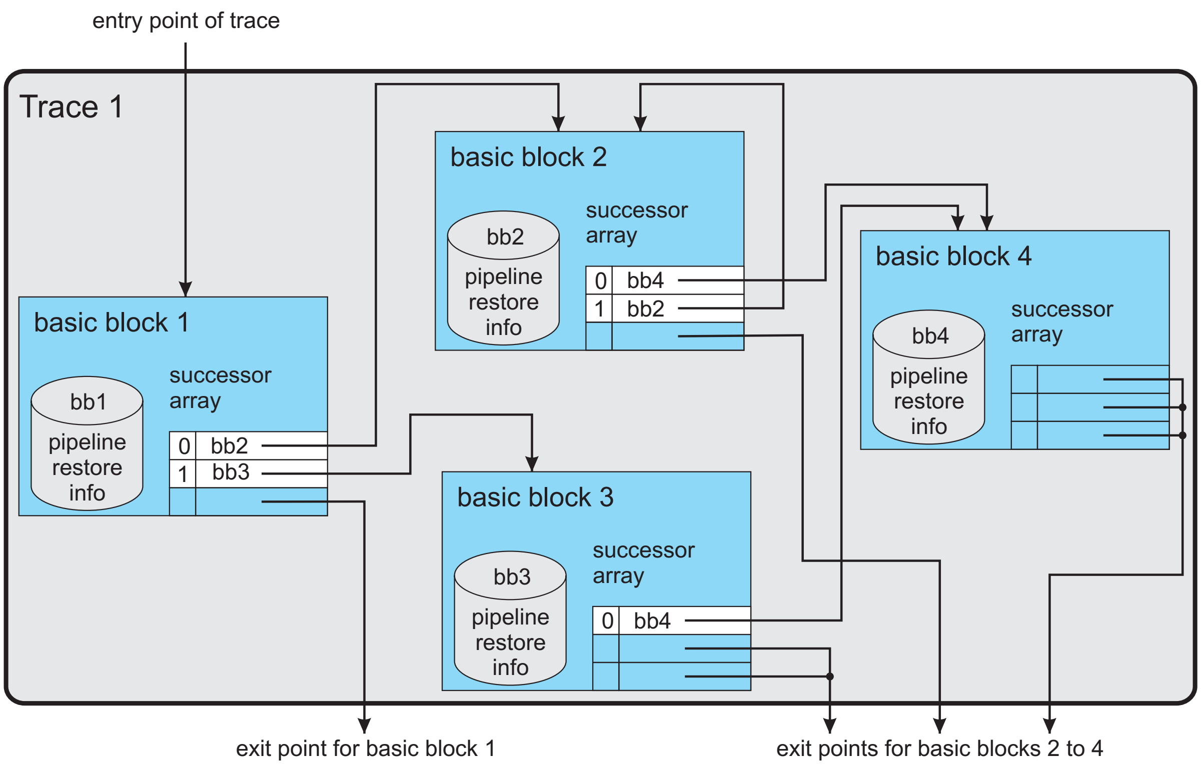


Figure 4.5: This trace combines four basic blocks and contains one loop.

4.9.1 LLVM Passes

The LLVM JIT framework allows to define different pass managers that combine a number of ordered optimisation passes. These pass managers are then used to transform the IR of LLVM *Function* objects. More passes might provide more efficient code but will need additional optimisation time. As a compromise we use two such pass managers to apply different levels of optimisations: basic optimisations are applied to basic blocks, more sophisticated passes are only applied to traces as they are executed more frequently.

Most passes for our passmanagers are selected to overcome obvious inefficiencies in the intermediate representation of the compilation unit. For example the most important optimisations for basic blocks are performed by the following (more or less self-explanatory) passes:

- ControlFlowGraphSimplification
- InstructionCombining
- DeadStoreElimination
- DeadInstructionElimination

- SparseConditionalConstantPropagation

All passes are used as black box and their internal functionality was not reviewed or modified. A list of available passes and a detailed description of the above passes can be found in [Lattner, 2002]. A systematic analysis of existent passes and additional user defined special purpose optimisation passes might improve the performance of compiler and/or binary blocks.

When using the LLVM JIT some mandatory additional optimisation and transformation passes are automatically applied to the intermediate representation to make executable code generation possible.

While efficient binary blocks are needed, compilation overhead has to be kept as low as possible. The most time consuming part of the compilation process is the instruction selection process. Therefore it is possible that the use of additional optimisations (that simplify the IR) lead to a shorter overall compiletime. Unfortunately compilation overhead in LLVM is fairly high compared to other JIT systems (see Section 5.3.2 and [Geoffray et al., 2008] for details).

4.9.2 Compiletime Constants

Besides LLVM optimisation passes we use compiletime constants to improve performance of binary blocks: Some register values and conditions are known to be invariant for each execution of a compiled block. For example the value of the program counter is identical each time a definite instruction at a given position in memory is fetched. We use this knowledge when building up the intermediate representation and omit loads from those registers.

Conditions that are known to be invariant are evaluated before compilation: For example the MIPS architecture has a constant register (register 0) and all writes to it have to be omitted. In the internal description a condition node is inserted to check the index of each register file write access. If this index is equal to zero the write access is not performed. Instead of generating a condition in the LLVM intermediate representation, which would involve the creation of additional host basic blocks, the condition is evaluated in the pipestage-function and code that writes to the registerfile is only generated for appropriate registerfile indices.

While most of those constant conditions would be optimised away by LLVM passes anyway, the compilation overhead is considerably reduced by this approach.

Another promising form of compiletime constants is the evaluation of active forward links at compiletime. For architectures that do not support conditional execution of non jump instructions (like MIPS) it is theoretically possible to evaluate the activeness of forwarding links at compiletime. This is even possible across borders of compilation units as special versions of blocks are generated depending on predecessor blocks and only the sequence of prior executed instructions is relevant for a forward to be active.

A not yet implemented analysis of the architecture would be needed to automatically activate this optimisation - its potential is shown in Section 5.6.1, where static evaluation of forwarding links was manually activated for the MIPS architecture.

A generic concept makes it easy to enable and disable these constant value optimisations: Each memory-element and each condition node may be marked as compiletime constant - implemented as a flag in the internal description. Memory-elements with this flag generate a LLVM constant in their *load()* method using their current (changeable) interpreter values. This is one of the reasons why compilation is done interleaved with the invocation of the interpreter. Condition nodes marked as compile constant are translated to if-else-statements in the code generating pipestage-functions - in this case all involved memory-elements are evaluated at compiletime (again using their current interpreter state).

4.9.3 Reduced Global Scope

The use of global variables for most of the memory-elements leads to some optimisation related considerations.

Some LLVM optimisation passes do not perform well when many load and store operations to global variables are performed. Furthermore, some global memory-elements have a reduced logical scope - their value is only relevant within the cycle it was written, which opens up additional optimisation potential.

A solution for both considerations is the use of local buffer variables to hold global values as illustrated in Figure 4.6: When the code for one basic block is generated, all occurrences of global variables are listed. For each of these global memory-elements a local copy is allocated and initialised at the top of the generated code. These caches are used instead of global variables within the whole basic block. After actual code generation, code that writes these caches back to the global variables is appended. While this rather complicated procedure seems counterproductive, it improves overall performance as some optimisation passes get faster or more effective (see Section 5.6).

Some variables in the internal description are used only within one clock cycle. For example a shadow memory-element is used to store values that are written to a register in the epilogue of a stage function. While this memory-element cannot be a local variable in the pipestage-function (as it would not be visible in the epilogue), it is not necessary to load its (global) value to the above described cache or write that cache back because a basic block does not end within a clock cycle - the cache can be used like a local variable then.

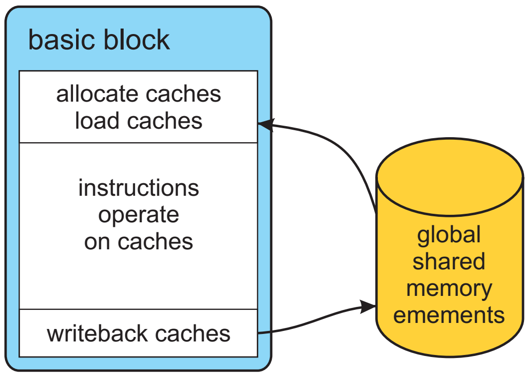


Figure 4.6: All accesses to global variables are done inside prologue and epilogue sections - the actual processing is done on local caches.

For the modelling of several architectural properties, like forwarding links, global memory-elements are used that are only relevant in the next cycle after they have been modified (but their values can be arbitrary in all following cycles and are ignored).

To account for these properties special scope attributes for global memory-elements are added to the internal description. They are divided into read and write properties and can be any combination of the attributes in Table 4.1.

Read	**Write**
SCOPE_READ_ALWAYS	SCOPE_WRITE_ALWAYS
SCOPE_READ_IF_FIRST	SCOPE_WRITE_IF_LAST
SCOPE_READ_NEVER	SCOPE_WRITE_NEVER

Table 4.1: Possible scope attributes for global memory-elements.

Global memory-elements with *READ_ALWAYS* and *WRITE_ALWAYS* attributes are handled as described above where no attributes are taken into account. The combination of *READ_NEVER* and *WRITE_NEVER* transforms the global memory-element to a local variable inside the generated code. And *READ_IF_FIRST* combined with WRITE_IF_LAST results in code that omits the initialisation, respectively writeback of local cache variables that are not used in the "borders" (first respectively last cycle) of a basic block.

Chapter 5

Evaluation

To evaluate the performance of the introduced LLVM based binary translation approach, we generated simulators for our two exemplary architectures. For each of them a set of benchmark applications is compiled and used to evaluate their behaviour. These applications are analysed regarding their translator concerning properties like basic block sizes or compilation overhead. A comparison between the interpreter and translator demonstrates the possibilities of dynamic binary translation for generated simulators at the structural level of the pipeline.

The main part of this chapter evaluates techniques and optimisations applied to the translator and their impact on overall simulation performance. Additionally simulator runtime is analysed to find good compromises between runtime and compile time.

While some of the presented results are promising it has to be noted that until now some architecture properties are not simulated in detail: particularly stalling of the pipeline and cache misses are not modelled. Therefore no dynamic timing behaviour is simulated which simplifies the translator framework as for example the state of caches have not to be taken into account. While simulated applications are correctly executed this leads to more or less significant inaccuracy of clock cycle counts (depending on the simulated architecture).

5.1 Benchmark Setup

The list of eleven applications used to test the generated simulators is a slightly modified subset of the *MiBench* embedded benchmark suite [Guthaus et al., 2001] plus one very simple, compute intensive prime number generator. A list of these applications and their runtime, in cycles of the source architecture, are shown in Table 5.1. Some of the **MiBench** applications were slightly adapted to either avoid unsupported system calls or to modify their runtime. The input files for some applications (marked with * in the table) are different for the two example architectures to account for different memory sizes - this manifests in incomparable runtimes for those applications.

	prime	**jpeg***	**crc32**	**sha**	**dijkstra**	**bitcount**
MIPS	20459	36151	645303	162018	350441	48
CHILI	435607	17815	1389091	278669	644530	42

	blowfish	**stringsearch**	**adpcm**	**gsm**	**rijndael**
MIPS	14877	8426	26623	109551	32619
CHILI	28305	11413	51285	136304	61327

Table 5.1: List of benchmark applications and simulation times in thousand cycles

The benchmarks for the MIPS R2000 architecture were compiled with *gcc version 3.4.6 (Gentoo 3.4.6-r2 p1.5, ssp-3.4.6-1.0, pie-8.7.10)*. For the CHILI 4-way VLIW architecture an experimental version of gcc (based on gcc 4.2.0) was used. All benchmarks were compiled with optimisation parameter *-O*. The influence of optimisation parameters on simulation performance is evaluated in Section 5.5.

Most benchmark applications need more cycles to complete on the VLIW architecture - which is remarkable as the four slots provide a much higher theoretical instruction per clock cycle ratio. This is mainly due to weak optimisations of the experimental compiler.

All tests were performed on a single core *AMD Athlon(tm) 64 Processor 3500+* with 2200MHz and 1GB of RAM running a 32-Bit Linux operating system. During the tests dynamic frequency scaling features of this processor were disabled to obtain constant duration of clock ticks, which were used to precisely measure execution time of different simulator components.

5.2 Comparison to Interpreter

Figures 5.1 and 5.2 compare the performance of interpreting and translating simulators for the MIPS respectively CHILI architecture. While the interpreter has more or less constant simulation speed in the range of 3.2 MHz respectively 0.7 MHz, the performance of the translator is very application-dependent.

The average simulation speed (AVG) indicates that the dynamic binary translator is over 10 times faster than the interpreter regarding the single slot MIPS architecture - and even more than 100 times speedup is reached for the CHILI VLIW architecture. CHILISIM a commercial product for cycle accurate simulation of the CHILI architecture is able to simulate at about 0.15 MHz when no cache misses are modelled. Compared to this simulator the translator can be over 1000 times faster (at least for some benchmarks).

In the next sections the determining application parameters that are responsible for the huge performance variations are explored.

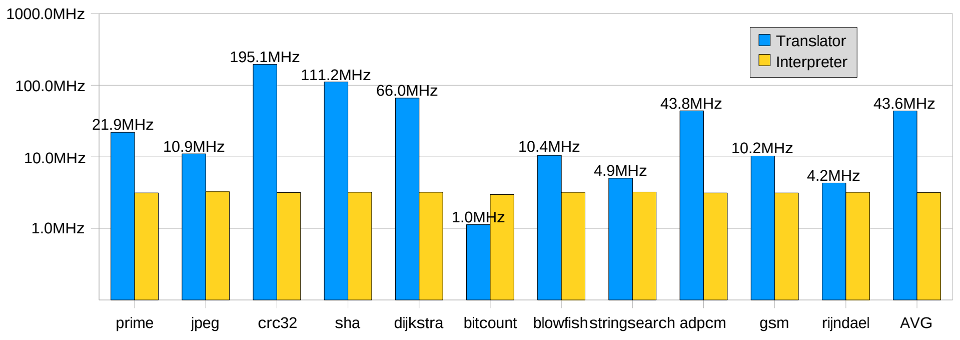


Figure 5.1: Interpreter versus translator, simulation speed in MHz for the MIPS architecture (logarithmic scale)

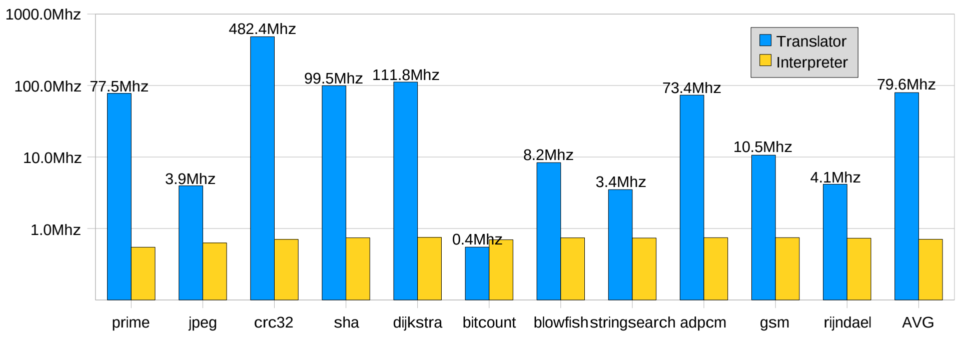


Figure 5.2: Interpreter versus translator, simulation speed in MHz for the CHILI architecture (logarithmic scale)

5.3 Analysis of Benchmark results

5.3.1 Benchmark Properties

Most notably, applications that only run for a few cycles like *bitcount* have reduced simulation speed as the overhead for execution profiling and compilation does not pay off. In fact this is the most important parameter and determines whether dynamic translation techniques can be successfully applied. Especially when a ”heavy weighted“ compilation infrastructure like LLVM is used, only ”long-running“ application benefit from dynamic translation. In our case long-running means applications that require more than about 500000 cycles to complete - of course this value depends on various parameters, like the source architecture or compilation threshold values.

Other important benchmark parameters are listed in Table 5.2. The average basic block sizes in clock cycles (as defined in Section 4.6.1) are listed for all compiled basic blocks and represent dynamic averages (blocks are weighted by their execution frequency). Basic blocks are terminated by jump instructions; the columns **Comp. Jumps** show static (non weighted) percentage of compiled blocks that end in a computed jump and cannot be linked to their successor blocks. On average about 6 percent of compiled blocks are duplicates of other blocks with different predecessors (column *duplicates* in table) - a result of the chosen way of handling overlapping instructions through multiple versions of basic blocks.

	MIPS			**CHILI**		
	duplicates	**Block Size**	**Comp. Jumps**	**duplicates**	**Block Size**	**Comp. Jumps**
prime	6%	3.3	10%	10%	6.1	10%
jpeg	7%	13.5	10%	2%	25.8	5%
crc32	5%	11.8	23%	3%	25.8	21%
sha	5%	18.3	16%	8%	21.7	14%
dijkstra	10%	7.8	13%	9%	14.6	14%
bitcount	0%	2.6	20%	-	-	-
blowfish	0%	81.6	6%	0%	169.9	13%
stringsearch	10%	5.2	10%	10%	8.9	11%
adpcm	20%	3.8	7%	3%	11.0	10%
gsm	7%	22.8	9%	4%	28.9	7%
rijndael	0%	17.1	12%	5%	31.7	9%
AVG	6.4%	*17.1*	*12%*	*5.4%*	*34.4*	*10%*

Table 5.2: Average dynamic block sizes, ratio of computed jumps and ratio of duplicates for compiled basic blocks.

In principal the simulator has three different techniques to simulate a clock cycle; these are: interpretation, execution in form of a compiled basic block and execution in a trace. The chosen compilation threshold values determine how all simulated cycles are distributed over these three possibilities (see Figures 5.3 and 5.4). Applications with large **trace** fractions (a consequence of long runtimes) are by trend executed faster. It is Remarkable that the characteristics of the benchmarks are nearly identical for the two architectures.

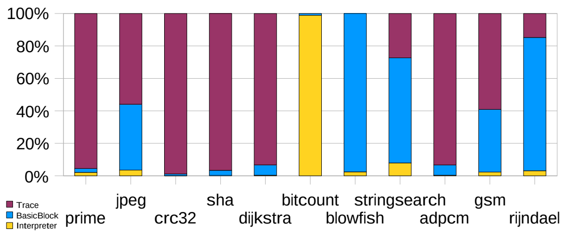


Figure 5.3: MIPS: Breakdown of cycles by used simulation technique.

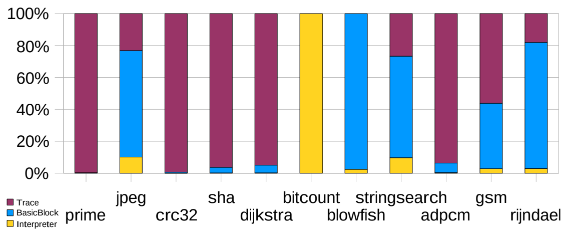


Figure 5.4: CHILI: Breakdown of cycles by used simulation technique.

5.3.2 Compile Time Overhead

The overall simulation time is divided into compilation-time and pure-runtime in Figures 5.5 and 5.6. The compilation threshold values for basic block and trace compilation are set to relatively high values for this evaluation: basic blocks are compiled after 600 executions and traces are formed after a block is executed more often than 40000 times. Even with those values, which result in only a few compiled blocks per application (see table Table 5.3), compilation often consumes more than a half of the overall simulation time.

The average compilation overhead for a single basic block is in the range of 20ms (MIPS) and 30ms (CHILI) and (approximately) linearly depends on basic block sizes. This is a rather high value compared to the average overall runtime of a few seconds. Therefore high compilation thresholds result in the best overall performance for a majority of the tested applications.

	MIPS				CHILI			
compile threshold	**6**	**60**	**600**	**6000**	**6**	**60**	**600**	**6000**
prime	235	235	102	5	169	169	49	10
jpeg	596	382	214	63	529	361	104	40
crc32	75	66	65	65	33	31	28	28
sha	163	119	94	34	129	99	74	44
dijkstra	574	515	428	212	402	373	300	144
bitcount	198	50	5	0	130	24	0	0
blowfish	174	46	18	8	184	48	15	8
stringsearch	276	267	227	62	249	238	204	57
adpcm	88	52	45	32	39	33	30	15
gsm	727	676	350	134	675	590	335	136
rijndael	214	173	99	90	194	156	95	87
AVG ratio of compiled blocks	*64%*	*49%*	*31%*	*14%*	*62%*	*48%*	*29%*	*14%*

Table 5.3: Number of compiled blocks for different compile threshold values

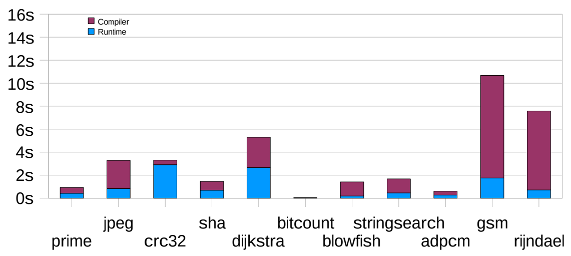


Figure 5.5: Compiletimes and overall runtimes for the MIPS architecture.

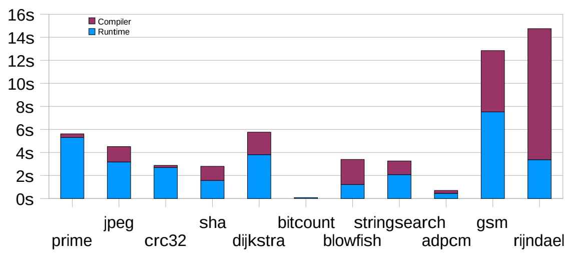


Figure 5.6: Compiletimes and overall runtimes for the CHILI architecture.

Of course long-running applications may need relatively less compilation overhead when all core loops of the application can be compiled into a few basic blocks - for example this is the case for the *crc32* benchmark.

To further confirm the tight correlation between the number of simulated cycles and overall performance, three applications for the MIPS architecture (applications for the

CHILI architecture have similar behaviour) were measured for different runtimes. To increase runtimes the inputset or the number of loop iterations for these applications was modified; their execution was interrupted after a certain number of cycles have been simulated. Figure 5.7 shows the simulation speed for different runtimes on a logarithmic scale - the dashed line represents the performance of the interpreter.

With longer runtime the performance converges to an applications specific maximal value which lies between 230MHz and 750MHz for these three applications (the **blowfish** benchmark for the CHILI architecture even reaches 875MHz when simulating 100,000M cycles). For a certain (short) runtime value, simulation speed is lowest because big parts of the compilation overhead do not pay off at this point (this value depends on the size of the applications kernel, which can roughly be estimated by Table 5.3).

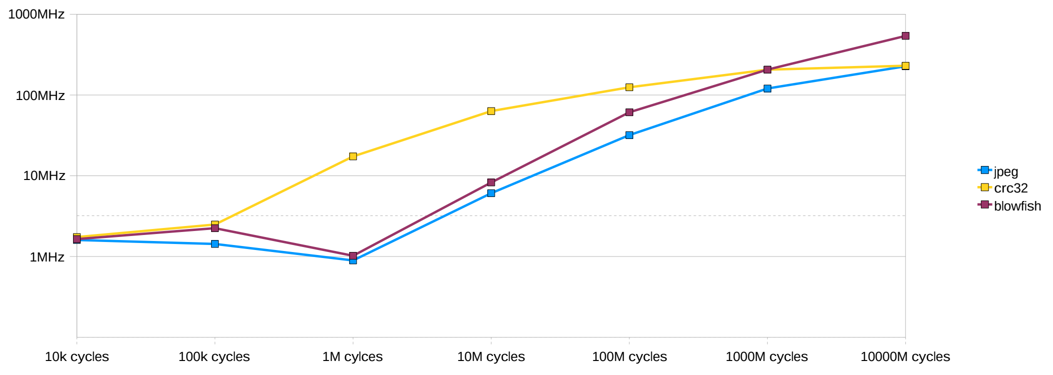


Figure 5.7: Performance for three applications with increasing simulation time

5.4 Trade-off for Compile Thresholds

To find a good compromise for compile thresholds several values were analysed. Figure 5.8 shows the performance for the values 6/400, 60/4000, 600/40000 and 6000/400000 which represent basic block/trace compile thresholds. Besides actual achieved simulation speed, the figure shows theoretical performance where compilation overhead is neglected. While theoretical performance is increasing for lower thresholds, average real performance has a maximum at a certain point (600/40000 for MIPS).

Benchmarks with larger basic block sizes like *blowfish* have higher theoretical speed because runtime efficiency of generated code for larger blocks is higher.

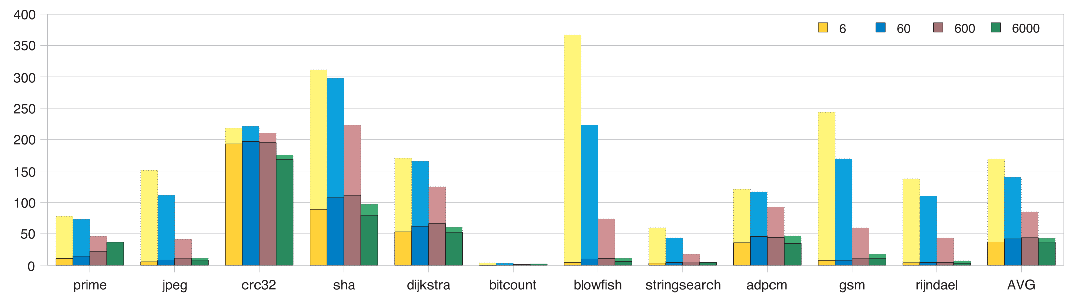


Figure 5.8: MIPS: simulation speed in MHz for different compilation thresholds with (solid box) and without (dashed box) compilation time.

5.5 Simulation of Unoptimised Applications

To uncover the impact of compiler optimisations an example source application was compiled with four different levels of optimisation. Figures 5.9, 5.10 and 5.11 show the results of the performed simulation runs. The idea behind this test was to check to which extent the translator is able to optimise applications at runtime. Results for *-O1*, *-O2* *-O3* are very similar; the unoptimised *-O0* version would need about the triple execution time on the source architecture (Figure 5.9) - its simulation only lasts about 60% longer (Figure 5.11). The better performance is however mostly because of longer simulation time and only partly benefits from the increased optimisation potential.

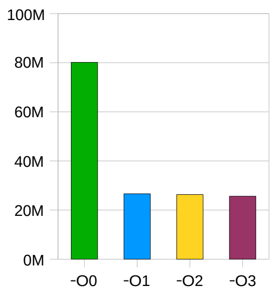


Figure 5.9: ADPCM: Simulated Cycles

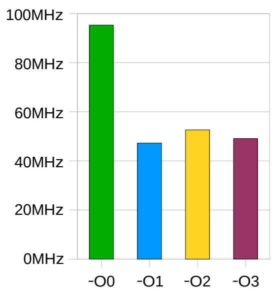


Figure 5.10: ADPCM: Simulation Speed

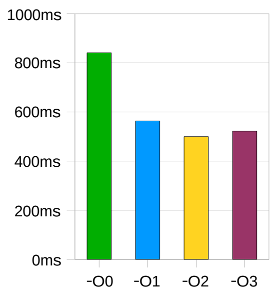


Figure 5.11: ADPCM: Overall Runtime

5.6 Impact of Applied Optimisations

In this section different optimisations, applied to the translating simulator are evaluated. First of all the connection between basic blocks is analysed. Three levels of block connections are compared: none, linking of blocks and formation of traces. The relative speedups of linking and traces (compared to no linking) are shown in Figures 5.12 and 5.13. On the one hand applications with high compiletime overhead do not benefit much form linking and may even be somewhat slower when additional overhead for trace compilation is needed. On the other hand linking of blocks and formation of traces definitely increases the execution speed of compiled blocks for all benchmarks. This results in significant speedups for applications with lower compilation overhead like *prime*.

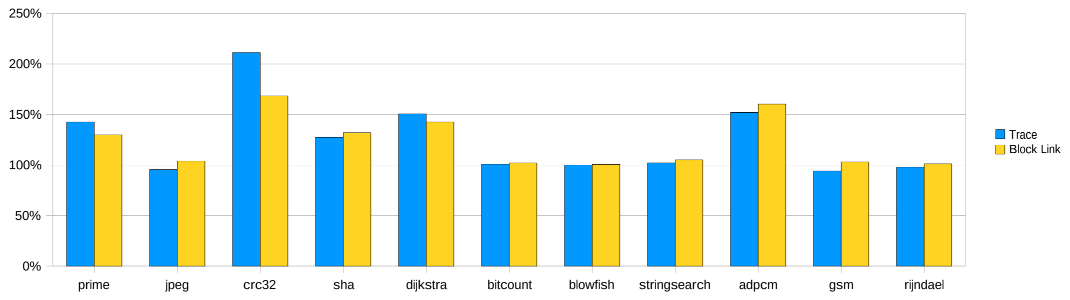


Figure 5.12: Impact of different block connections for the MIPS architecture

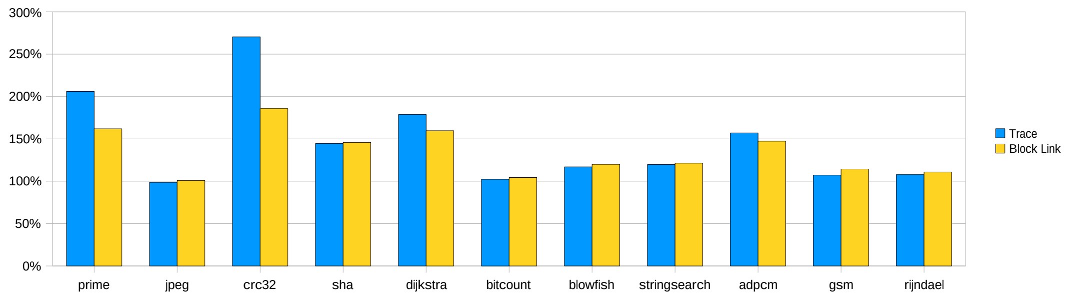


Figure 5.13: Impact of different block connections for the CHILI architecture

Traces only moderately improve simulation performance (in addition to block linking); this has several reasons. Traces are only compiled after longer runtime and the local cache optimisations reduce trace optimisation potential. As mentioned earlier, LLVM optimisation passes do not perform well when global variables are used. But values inside a Trace are passed by global values: Traces consist of the inlined contents of several basic block functions - these functions however eventually operate on global variables (even with scope optimisations they access the global state at least at begin and end of the basic block).

The big influence of local caches and scope optimisations is shown in Figures 5.14 and 5.15. All values are relative speed comparisons to a simulator that has local caches deactivated. The use of local caches alone increases the average simulation performance by about 50% percent - mainly because of faster compilation and lower LLVM optimisation overhead.

When special scope reduction techniques as described in Section 4.9.3 are used, the compilation overhead is further reduced for most benchmarks. (An exception is *rijndael* which has about 20 percent increased compilation time.) At the same time the eventually generated host code gets more effective as many unnecessary read and write operations to global variables are avoided.

When the simulator is run without active LLVM passes the effectiveness of generated code is decreased because no optimisations are made - but even worse the overall compilation requires much more time in this case. The reason for this is slow instruction selection, which consumes more time when big (unoptimised) intermediate representation has to be translated to host code. The runtime of optimisation passes is nearly negligible in comparison.

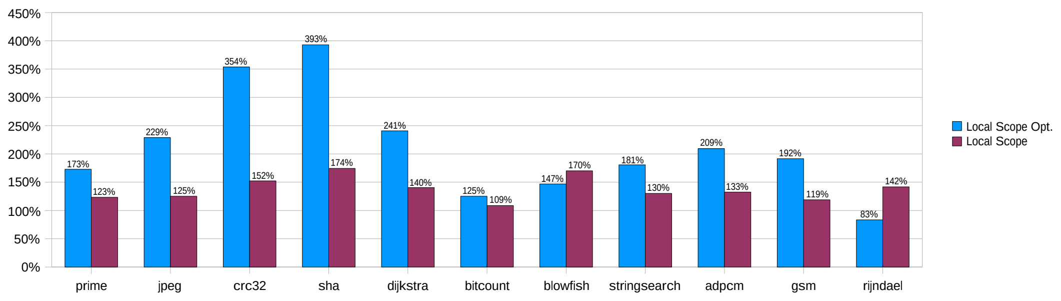


Figure 5.14: MIPS: Relative performance improvements gained by the use of local caches

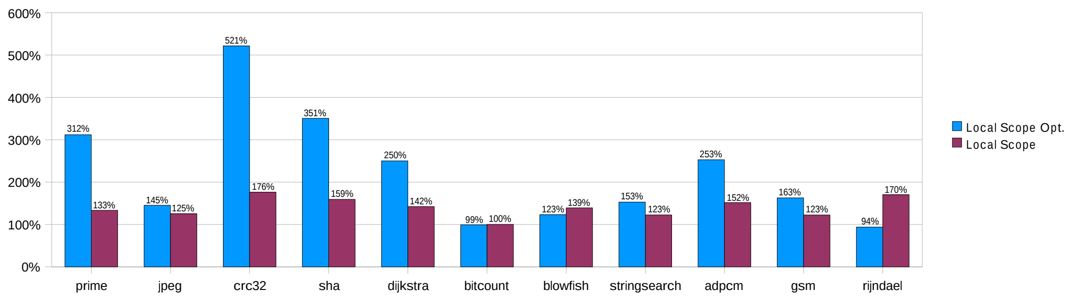


Figure 5.15: CHILI: Relative performance improvements gained by the use of local caches

The backend of the LLVM JIT framework is the performance bottleneck of the translation process and has presumably much potential to be improved. Therefore a detailed analysis of single optimisation passes is intentionally omitted; the applied optimisation passes are now selected to minimise instruction selection time.

5.6.1 Constant Evaluation of Forward Links

The last Figure (5.16) shows the influence of compile time evaluation of active forward links as described in Section 4.9.2. This optimisation can only be applied for architectures where no conditional register write instruction exists. This optimisation was not enabled for any of the previous evaluations as it has to be enabled manually - an automated analysis that identifies architectures without conditional instructions is not yet implemented. While the MIPS architecture benefits from this optimisation, the CHILI simulator will produce incorrect results if it is applied. As virtually each instruction in the MIPS architecture has to check its source operands for active forward links the improvements of this optimisation are relatively large.

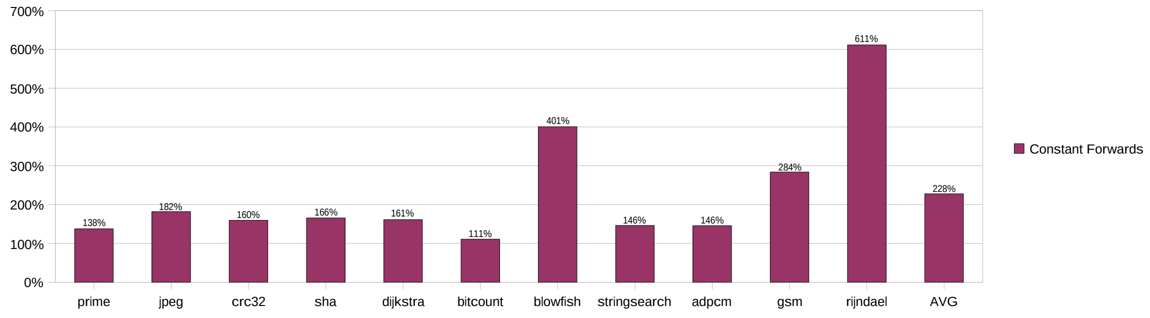


Figure 5.16: Performance speedups of MIPS simulator with static compiletime evaluation of active forwarding links

Conclusion

An approach to (almost) automatically generate a full functional architecture simulator that uses dynamic binary translation to achieve high performance simulation on the structural processor-pipeline level was presented.

The implementation in form of a module for the xADL framework enables the use of a well defined compact structural description languages that can be shared with other modules that generate compilers or hardware descriptions. Preprocessing of architecture descriptions is done by this framework, which has significantly reduced the implementation effort for the presented simulator generator.

LLVM as just in time compilation framework brings flexibility and enables runtime generation of highly optimised code for different host architectures - the downside is its high compilation overhead. Preparatory optimisations that are applied to the compilation unit before it is handed over to LLVM can reduce this disadvantage to some degree and furthermore enhance the efficiency of the eventually generated host code.

High compilation overhead forces the simulator to only translate very frequently executed blocks and to rely on interpretation for all others. This makes the translator inefficient for short benchmarks - nonetheless they can be simulated in reasonable time (in the order of a few seconds for the tested architectures). In situations where simulation performance is most important, because applications with long runtimes have to be simulated, the translator taps its full potential and is able to improve simulation performance up to three orders of magnitude (compared to interpretation).

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Wien, am 30. September 2008

David Rigler

Bibliography

[Asanovic, 2000] Asanovic, K. (2000). Energy-exposed instruction set architectures.

[Aycock, 2003] Aycock, J. (2003). A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113.

[Azevedo et al., 2005] Azevedo, R., Rigo, S., Bartholomeu, M., Araujo, G., Araujo, C., and Barros, E. (2005). The archc architecture description language and tools. *Int. J. Parallel Program.*, 33(5):453–484.

[Bala et al., 2000] Bala, V., Duesterwald, E., and Banerjia, S. (2000). Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA. ACM.

[Bartholomeu et al., 2003] Bartholomeu, M., Rigo, S., Azevedo, R., and Araujo, G. (2003). Emulating operating system calls in retargetable ISA simulators. Technical Report IC-03-29.

[Brandner et al., 2007] Brandner, F., Ebner, D., and Krall, A. (2007). Compiler generation from structural architecture descriptions. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 13–22, New York, NY, USA. ACM.

[Braun et al., 2004] Braun, G., Nohl, A., Sheng, W., Ceng, J., Hohenauer, M., Scharwächter, H., Leupers, R., and Meyr, H. (2004). A novel approach for flexible and consistent adl-driven asip design. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 717–722, New York, NY, USA. ACM.

[Cmelik and Keppel, 1994] Cmelik, R. and Keppel, D. (1994). Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137.

[D'Errico and Qin, 2006] D'Errico, J. and Qin, W. (2006). Constructing portable compiled instruction-set simulators: an adl-driven approach. In *DATE '06: Proceedings of* *the conference on Design, automation and test in Europe*, pages 112–117, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

[Fauth et al., 1995] Fauth, A., Praet, J. V., and Freericks, M. (1995). Describing instruction set processors using nml. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA. IEEE Computer Society.

[Fellnhofer, 2008] Fellnhofer, A. (2008). Master's thesis, Institut für Computersprachen der Technischen Universität Wien.

[Fellnhofer and Rigler, 2006] Fellnhofer, A. and Rigler, D. (2006). iboy- fast system level emulation using dynamic binary translation (a nintendo gameboy emulation system for apple ipods).

[Geoffray et al., 2008] Geoffray, N., Thomas, G., Clément, C., and Folliot, B. (2008). A lazy developer approach: Building a jvm with third party software. In *International Conference on Principles and Practice of Programming In Java (PPPJ 2008)* , Modena, Italy.

[Gorjiara et al., 2006] Gorjiara, B., Reshadi, M., Chandraiah, P., and Gajski, D. (2006). Generic netlist representation for system and pe level design exploration. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 282–287, New York, NY, USA. ACM.

[Guthaus et al., 2001] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA. IEEE Computer Society.

[Hadjiyiannis et al., 1997] Hadjiyiannis, G., Hanono, S., and Devadas, S. (1997). ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302.

[Hadjiyiannis et al., 1999] Hadjiyiannis, G., Russo, P., and Devadas, S. (1999). A methodology for accurate performance evaluation in architecture exploration. In *Design Automation Conference*, pages 927–932.

[Hennessy and Patterson, 2007] Hennessy, J. L. and Patterson, D. A. (2007). *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 4. edition.

[Lattner, 2002] Lattner, C. (2002). LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. *See* `http://llvm.cs.uiuc.edu`.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA. IEEE Computer Society.

[Levine et al., 1992] Levine, J., Mason, T., and Brown, D. (1992). *lex & yacc, 2nd Edition (A Nutshell Handbook).* O'Reilly.

[Luk et al., 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200.

[Mishra, 2005] Mishra, N. P. (2005). Architecture description languages for programmable embedded systems. In *IEE Proceedings: Computers and Digital Techniques*, volume 152, pages 285–297.

[Mishra and Dutt, 2004] Mishra, P. and Dutt, N. (2004). Modeling and validation of pipeline specifications. *Trans. on Embedded Computing Sys.*, 3(1):114–139.

[Mishra et al., 2004] Mishra, P., Kejariwal, A., and Dutt, N. (2004). Synthesis-driven exploration of pipelined embedded processors. In *VLSID '04: Proceedings of the 17th International Conference on VLSI Design*, page 921, Washington, DC, USA. IEEE Computer Society.

[Pees et al., 1999] Pees, S., Hoffmann, A., Zivojnovic, V., and Meyr, H. (1999). Lisa—machine description language for cycle-accurate models of programmable dsp architectures. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 933–938, New York, NY, USA. ACM.

[Pesch and Osier, 1993] Pesch, R. H. and Osier, J. M. (1993). The gnu binary utilities.

[Probst, 2002] Probst, M. (2002). Dynamic binary translation. In *UKUUG Linux Developer's Conference 2002.*

[Qin and Malik, 2002] Qin, W. and Malik, S. (2002). *Architecture Description Languages for Retargetable Compilation, in The Compiler Design Handbook: Optimizations & Machine Code Generation.* CRC Press.

[Qin et al., 2004] Qin, W., Rajagopalan, S., and Malik, S. (2004). A formal concurrency model based architecture description language for synthesis of software development tools. *SIGPLAN Not.*, 39(7):47–56.

[Rajesh and Moona, 1999] Rajesh, V. and Moona, R. (1999). Processor modeling for hardware software codesign. In *VLSID '99: Proceedings of the 12th International Conference on VLSI Design - 'VLSI for the Information Appliance'*, page 132, Washington, DC, USA. IEEE Computer Society.

[Scott et al., 2003] Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., and Soffa, M. L. (2003). Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 36–47, Washington, DC, USA. IEEE Computer Society.

[Seng et al., 2002] Seng, S. P., Palem, K. V., Rabbah, R. M., Wong, W. F., Luk, W., and Cheung, P. Y. K. (2002). *PD-XML: extensible markup language for processor description*, pages 437–440.

[Shi et al., 2007] Shi, H., Wang, Y., Guan, H., and Liang, A. (2007). An intermediate language level optimization framework for dynamic binary translation. *SIGPLAN Not.*, 42(5):3–9.

[Sohi, 1990] Sohi, G. S. (1990). Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. Comput.*, 39(3):349–359.

[Topham and Jones, 2007] Topham, N. and Jones, D. (2007). High speed cpu simulation using jit binary translation. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation.*

[Ung and Cifuentes, 2000] Ung, D. and Cifuentes, C. (2000). Machine-adaptable dynamic binary translation. *SIGPLAN Not.*, 35(7):41–51.

[Ung and Cifuentes, 2001] Ung, D. and Cifuentes, C. (2001). Optimising hot paths in a dynamic binary translator. *SIGARCH Comput. Archit. News*, 29(1):55–65.